Function Reference Guide

c-tree Plus™

c-tree Plus™ File Handler

Function Reference Guide

Sixth Edition

For c-tree Plus™ Version 6.0 or later,
the FairCom® Server,
and
the FairCom® SQL Server

©1984, 1986, 1987, 1988, 1990 FairCom Corporation

ALL RIGHTS RESERVED

Published by

FairCom Corporation
4006 West Broadway
Columbia, MO 65203
(314) 445-6833

Copyright © 1984, 1986, 1987, 1988, 1989, 1990 FairCom Corporation

All rights reserved. No part of this publication may be stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior written permission of FairCom Corporation. Printed in the United States of America

First Printing: October 1990

c-tree, c-tree Plus, r-tree, d-tree, the c-tree circular disk logo, and FairCom are trademarks of FairCom Corporation.

UNIX is a trademark of AT&T

MS-DOS, OS/2 and XENIX are trademarks of Microsoft

Macintosh is a trademark licensed to Apple Computer Co.

IBM is a trademark of International Business Machines Corp.

c-tree Plus™ Function Reference Guide
Table of Contents

c-tree Plus™ Functions

1. c-tree Plus™ Function Descriptions

Abort 1-3
AddKey 1-4
AddRecord 1-6
AddResource 1-8
AddVRecord 1-10
AllocateSet 1-12
Begin 1-13
ChangeSet 1-15
ClearTranError 1-16
CloseFile 1-17
CloseIFile 1-18
CloseISAM 1-19
CloseRFile 1-20
Commit 1-21
cpybuf 1-23
CreateDataFile 1-24
CreateDataFileXtd 1-27
CreateIFile 1-28
CreateIFileXtd 1-30
CreateIndexFile 1-31
CreateIndexFileXtd 1-34
CreateIndexMember 1-35
CreateISAM 1-37
CreateISAMXtd 1-40
CurrentFileOffset 1-42
CurrentISAMKey 1-43
DeleteFile 1-44
DeleteKey 1-45
DeleteKeyBlind 1-47
DeleteRecord 1-49
DeleteResource 1-51
DeleteVRecord 1-53
DoBatch 1-55
EnableResource 1-59
EstimateKeySpan 1-60

FirstInSet 1-61
FirstKey 1-63
FirstRecord 1-64
FreeSet 1-66
GetAlternateSequence 1-67
GetDODA 1-68
GetFileInfo 1-70
GetGTEKey 1-72
GetGTERecord 1-74
GetGTKey 1-76
GetIFile 1-77
GetKey 1-78
GetLTEKey 1-79
GetLTERecord 1-80
GetLTKey 1-82
GetRecord 1-83
GetResource 1-85
GetSerialNbr 1-88
GetSymbolicNames 1-89
GetTempFileName 1-90
InitCTree 1-91
InitCTreeXTD 1-93
InitISAM 1-95
InitISAMXtd 1-97
KeyAtPercentile 1-99
LastInSet 1-101
LastKey 1-103
LastRecord 1-104
LoadKey 1-106
LockData 1-108
LockISAM 1-110
NewData 1-114
NewVData 1-116
NextInSet 1-118
NextKey 1-120
NextRecord 1-121
NbrOfKeyEntries 1-123
NbrOfRecords 1-124
OpenFile 1-125
OpenFileWithResource 1-127
OpenFileWithResourceXtd 1-129
OpenFileXtd 1-130
OpenIFile 1-131
OpenIFileXtd 1-133
OpenISAM 1-134
OpenISAMXtd 1-137
PositionSet 1-139
PreviousInSet 1-141
PreviousKey 1-143

PreviousRecord 1-144
PutDODA 1-146
ReadData 1-147
ReadVData 1-148
RebuildIFile 1-150
RebuildIFileXtd 1-153
ReleaseData 1-155
ReleaseVData 1-157
ReReadRecord 1-158
ReReadVRecord 1-159
ResetRecord 1-161
RestoreSavePoint 1-163
ReWriteRecord 1-165
ReWriteVRecord 1-168
Security 1-171
SetAlternateSequence 1-173
SetRecord 1-174
SetSavePoint 1-176
SetVariableBytes 1-178
StopUser 1-180
TransformKey 1-181
UpdateFileMode 1-183
UpdateResource 1-185
VDataLength 1-187
VRecordLength 1-188
WriteData 1-189
WriteVData 1-190

SQL Functions/Statements

2. SELECT Statement

Simple SELECT 2-1
The Authorization Identifier Prefix 2-1
SELECT with Condition 2-2
SELECT with DISTINCT - Eliminating Duplicate Rows 2-2
SELECT with Two Tables 2-2
SELECT with ANY 2-3
VALUE List 2-4
SELECT with IN 2-4
SELECT with Three Levels 2-4
SELECT with Interblock Reference 2-4
Interblock Reference - Same Table in Both SELECTS 2-5
SELECT with ALL 2-5
SELECT with Comparison Operator and Sub-Query 2-5
SELECT with EXISTS 2-5
SELECT with Computed Values 2-6
SELECT with ANSI Functions 2-6
SELECT with String Functions 2-8

SELECT with LENGTH Function 2-8
SELECT with RTRIM Function 2-8
SELECT with RPAD Function 2-8
SELECT with LOWER Function 2-9
SELECT with UPPER Function 2-9
SELECT with SUBSTR Function 2-9
SELECT with CONCAT Function 2-10
SELECT with DATE/TIME FUNCTIONS 2-10
SELECT with TRUNCATE Function 2-10
SELECT with TO_DATE Function 2-10
SELECT with TO_CHAR Function 2-11
SELECT with Time Addition 2-11
SELECT with Time Difference 2-11
SELECT with DATE_REMAINDER 2-12
SELECT with BETWEEN 2-13
SELECT with LIKE 2-13
SELECT with GROUP BY 2-14
SELECT with GROUP BY and HAVING 2-14
SELECT with ORDER BY 2-14
ANSI ORDER BY Features 2-14
ORDER BY with NODUP 2-15
ORDER BY with BREAK 2-16
ORDER BY with COMPUTE 2-16
SELECT with AT END REPORT 2-16
Report Generation Enhancements 2-17
Output Formatting 2-17
Pattern String 2-17
Template String 2-19
NODUP Modifier 2-20
Column Heading Override 2-20
Column Justification 2-21
Wrapping 2-21
Summary of Column Specification Features 2-21
The SELECT INTO Statement 2-21

3. SQL / C Interface Function Reference

destbuf 3-3
destfil 3-4
r_fetch 3-5
r_insert 3-6
r_update 3-7
setdest 3-8
setvars 3-9
sql1 3-10
sql2 3-11

4. SQL Statement Reference

Notational Conventions 4-1

ALTER TABLE 4-3
CLOSE 4-4
COMMIT 4-5
CREATE INDEX 4-6
CREATE SCHEMA 4-7
CREATE TABLE 4-8
CREATE VIEW 4-11
DECLARE CURSOR 4-13
DELETE 4-14
DROP TABLE 4-15
DROP VIEW 4-16
FETCH 4-17
FETCHP 4-18
GRANT 4-19
INSERT INTO 4-22
Standard 4-22
With Select 4-22
OPEN 4-24
REVOKE 4-25
ROLLBACK WORK 4-26
SELECT 4-27
SET AUTHORIZATION 4-28
UPDATE 4-29

Combined Index

c-tree Plus™ Functions

All of the functions that are used in all modes of c-tree Plus, with the exception of the SQL functions, are listed alphabetically in this section.

1. c-tree Plus™ Function Descriptions

These c-tree Plus function definitions are presented in alphabetical order, each starting on a new page, to facilitate your search for information.

Please Note: any references to pages in the c-tree Programmer's Reference Guide will be prefaced with "PRG", as in "See page PRG 5-1".

Function definitions are comprised of the following sections:

- **Short Name** Internal c-tree function name (see Appendix PRG C)
- **Type** ISAM function or low level function
- **Declaration** function and parameter declarations
- **Description** detailed description
- **Return** explanation of return values
- **Example** programming example
- **Limitations** restrictions on use and/or shortcomings
- **See Also** related functions and information

Recall that after each c-tree Plus function call, a global error code variable will contain the current error code (or zero if no error). In the case of low level file functions, the error code variable is *uerr_cod*; for the ISAM functions, the variable is *isam_err*. In addition, the ISAM functions also set the variable *isam_fil* with the data or index file number involved.

The error codes listed in each RETURN section of the function descriptions are not exhaustive. The only error codes included are those which have a meaningful context for the function in question.

NOTE: Those functions that are unique to the FairCom SQL Server are described separately in Chapters 4 and 3

Abort

Abort the current transaction in progress.

SHORT NAME TRANABT

TYPE Low level data file function

DECLARATION

```
COUNT Abort();
```

DESCRIPTION

Abort will abort the current transaction. All file update actions since the last **Begin** call will not be committed. All known locks are released, and the transaction is ended.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	no error occured
71	TNON_ERR	there is no active transaction pending.

EXAMPLE

```
COUNT savepoint;
VOID domaster()
{Begin(ENABLE|TRNLOG);                    /*start transaction with locks*/
 while(another());                        /* get next record to add */
 { savepoint = SetSavePoint();            /* get save point at beginning
                                             of each master record */
   if (add_master()<0) Abort();           /*abort if can't add master rec.*/
   dodetail();                            /* process detail records */
 }
 if (Commit(FREE)>0) printf("\nError %d in transaction",uerr_cod);
 return;
}
VOID dodetail()
{while(moredetail());                     /* get next detail record to add */
 { if (add_detail()<0)                    /* add details, if possible */
   { RestoreSavePoint(savepoint) /* with error, return to savept*/
     return;
   }
 }
}
```

SEE ALSO **Begin, Commit, ClearTranError.**

AddKey

Add key value to index file

SHORT NAME ADDKEY

TYPE Low level index file function

DECLARATION

```
COUNT AddKey(keyno,target,recbyt,typadd)
COUNT   keyno;
VOID    *target;
LONG    recbyt;
COUNT   typadd;
```

DESCRIPTION **AddKey** inserts the key value pointed to by *target* and the associated data record position *recbyt* into index file number *keyno*. *typadd* determines how a full B-Tree node is split. The symbolic constants (defined in ctport.h) REGADD, INCADD and DECADD represent the three node splitting options:

typadd Value	Symbolic Constant	Action
0	REGADD	Split nodes in half. This is the ordinary mode for random key value insertions.
1	INCADD	Uneven node split to accommodate insertions made in ascending key value order. For example, if the key values are already sorted in ascending order, using INCADD for the *typadd* parameter will create more compact B-Trees.
2	DECADD	Uneven node split to accommodate insertions made in descending key value order.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful key value insertion.
2	KDUP_ERR	Target key value already in index. No insertion performed.
28	ZDRN_ERR	*recbyt* is zero. No insertion performed.
39	FULL_ERR	No more B-Tree nodes available. It is necessary to rebuild index.

EXAMPLE

```
TEXT        target[11];
LONG        recbyt;
COUNT       name_key;

if   (AddKey(name_key,target,recbyt,REGADD))
   printf("\nAddKey error = %d",uerr_cod);
```

LIMITATIONS

AddKey does not pad key values. Therefore, if you pass a key value which is not completely specified to its full length, **AddKey** will effectively pad the value with whatever "garbage" follows the key value in memory.

Key values added to an index which supports duplicate keys (*keydup* = 1 in **CreateIndexFile** or **CreateIndexMember**) have their last four bytes overwritten by the associated data record position. In this way, identical key values become distinguishable.

SEE ALSO

CreateIndexFile and **CreateIndexMember**: the key value characteristics of length, type (left or right scan), and allowance for duplicate values are specified at the time the index file is created.

AddRecord

Add a new fixed length data record and its corresponding key values to the ISAM files

SHORT NAME

ADDREC

TYPE

ISAM function fixed length record

DECLARATION

```
COUNT AddRecord(datno,recptr)
COUNT datno;
VOID  *recptr;
```

DESCRIPTION

AddRecord adds the fixed length data record pointed to by *recptr* to data file number *datno*. It automatically adds the key values defined for the data file to their respective index files. If successful, the new data record becomes the current ISAM record for data file datno.

In multi-user applications **AddRecord** automatically causes a data record lock to be placed on the new record. Therefore, **LockISAM(ENABLE)** and **LockISAM(FREE)** should be called before and after **AddRecord**. However, if you are using transaction processing you will usually enable and free locks during the **Begin** and **Commit** functions.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful addition of a new data record.
2	KDUP_ERR	Duplicate key value found in index file number *isam_fil*. Data record not added and no key values inserted.
31	DELFLG_ERR	Attempt to re-use data record whose first byte is not set to ff (hex). Either the data file is corrupted, in which case you should rebuild the ISAM files, or the C255 constant defined in *ctcmpl.h* is not set properly.
33	DNUL_ERR	*recptr* is null. No action taken.
37	WRITE_ERR	Most likely disk or directory is full. Files are left in an indeterminate state and should be rebuilt.
48	FMOD_ERR	*datno* is assigned to a variable record length file.
199	NSCH_ERR	key segment refers to schema, but schema not defined.
433	SSCH_ERR	segment definition inconsistent with schema.

Value	Symbolic Constant	Explanation
445	SDAT_ERR	not enough data to assembly key.
446	BMOD_ERR	invalid key mode.

EXAMPLE

```
COUNT          invfil;
struct invd {
   TEXT        delflg;
   LONG        part_no;
   TEXT        part_name[60];
   LONG        on_hand;
} recbuf;

scanf("%ld %59s %ld",&recbuf.part_no,recbuf.part_name,
   &recbuf.on_hand);
recbuf.delflg = '\0'; /* clear delete flag */
if (LockISAM(ENABLE) || AddRecord(invfil,&recbuf) ||
     LockISAM(FREE))
   printf("\nAddRecord error %d in file %d",
           isam_err,isam_fil);
```

SEE ALSO

Chapter PRG 5 discusses how the relationship among the data files and their corresponding index files is specified.

AddResource

Add a Resource to a data file.

SHORT NAME ADDRES

TYPE Resource function

DECLARATION

```
COUNT AddResource(filno, resptr, varlen)
COUNT  filno;
VOID  *resptr;
VRLEN  varlen;
```

DESCRIPTION **AddResource** is used to add a Resource to a file *filno*. Every Resource is identified by a unique combination of a Resource Type and a Resource Number. The Resource Number can optionally be assigned by c-tree during the call to **AddResource**. In addition, each Resource can be identified by a Resource Name. The Resource Name is **not** guaranteed to be unique.

resptr points to a resource data block. This block contains the following fields:

Byte Offset	Data Type	
0 - 3	unsigned long integer	Resource Type
4 - 7	unsigned long integer	Resource Number
8 - m	null terminated character string	Resource Name
m + 1 - n	any collection of data types desired	Resource Information

The Resource Type must be a value greater than 65536.

If the Resource Number is a value of 0xffffffffL then c-tree will assign the next available Resource Number for this Resource Type in the specified data file to this Resource. **AddResource** will place the assigned Number in the Resource Data Block.

The Resource Name is optional. c-tree does not guarantee unique Resource Names. Do not use names that start with "FC!" or "RD!", as these are already reserved by FairCom.

The Resource Information is any collection of data that you wish to store as a Resource. It can be a character string, a structure, or any variable type.

The parameter *varlen* specifies the length of the entire Resource Data Block.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful addition of a new Resource.
37	WRITE_ERR	Most likely disk or directory is full. Files are left in an indeterminate state and should be rebuilt
147	VDLFLG_ERR	Attempt to re-use a data record which is still active. Apparently the data file is corrupted in which case you should rebuild the file.
401	RNON_ERR	Resources have not been enabled for this data file.
403	RZRO_ERR	Empty Resource ID.
404	RDUP_ERR	Resource ID already added. There must be a unique combination of Type and Number.

EXAMPLE

```
#include "ctstdr.h"
#include "ctdecl.h"

structure {                              /* Resource Data Block structure */
   ULONG  resource_type;
   ULONG  resource_number;
   TEXT   variable_info[1016];    * This will hold the Resource
                                    Name and Data*/
} my_resource;

my_resource.resource_type    = 0x10001L;      /* 65537 */
my_resource.resource_number  = 0xffffffffL;   /* c-tree to assign
                                                 number*/
strcpy(my_resource.variable_info,"M&!resource");
strcat(my_resource.variable_info,"Actual Resource Data");

if (AddResource(filno,&my_resource,
   (VRLEN)(8 + strlen(my_resource.variable_info))) == 0 )
{
        printf("\nThe resource has been assigned number %ld",
             my_resource.resource_number);
        printf("\nThe resource is located at byte offset %ld.",
             GetResource(filno,"MY!resource",&my_resource,1024L);
} else
        printf("\nCould not add resource. Error #%d",uerr_cod);
```

SEE ALSO **GetResource, UpdateResource, DeleteResource**

AddVRecord

Add a new variable length data record and its corresponding key values to the ISAM files

SHORT NAME ADDVREC

TYPE ISAM function variable length record

DECLARATION

```
COUNT AddVRecord(datno,recptr,varlen)
COUNT  datno;
VOID  *recptr;
VRLEN  varlen;
```

DESCRIPTION

AddVRecord adds the data record with length in bytes given by *varlen* and pointed to by *recptr* to data file number *datno*. It automatically adds the key values defined for the data file to their respective index files. If successful, the new data record becomes the current ISAM record for data file *datno*.

In multi-user applications AddVRecord automatically causes a data record lock to be placed on the new record. Therefore, **LockISAM(ENABLE)** and **LockISAM(FREE)** should be called before and after **AddVRecord**. However, if you are using transaction processing you will usually enable and free locks during the **Begin** and **Commit** functions.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful addition of a new data record.
2	KDUP_ERR	Duplicate key value found in index file number *isam_fil*. Data record not added and no key values inserted.
33	DNUL_ERR	*recptr* is null. No action taken.
37	WRITE_ERR	Most likely disk or directory is full. Files are left in an indeterminate state and should be rebuilt.
48	FMOD_ERR	*datno* is assigned to a fixed record length file.
147	VDLFLG_ERR	Attempt to re-use a data record which is still active. Apparently the data file is corrupted in which case you should rebuild the ISAM files.
149	VRLN_ERR	*varlen* is less than the minimum record length for this variable length file.
199	NSCH_ERR	key segment refers to schema, but schema not defined.

Value	Symbolic Constant	Explanation
433	SSCH_ERR	segment definition inconsistent with schema.
445	SDAT_ERR	not enough data to assembly key.
446	BMOD_ERR	invalid key mode.

EXAMPLE

```
COUNT          invfil;
struct invd {
   LONG        part_no;
   LONG        on_hand;
   TEXT        part_name[120];
} recbuf;

scanf("%ld %ld %119s",&recbuf.part_no,&recbuf.on_hand,
   recbuf.part_name);
if (LockISAM(ENABLE) || AddVRecord(invfil,&recbuf,9 +
   strlen(recbuf.part_name)) || LockISAM(FREE))
     printf("\nAddVRecord error %d in file %d",isam_err,
         isam_fil);
```

SEE ALSO

Chapter PRG 5 discusses how the relationship among the data files and their corresponding index files is specified.

AllocateSet

Allocate buffer space for multiple sets.

SHORT NAME ALCSET

TYPE ISAM function

DECLARATION

```
COUNT AllocateSet(numset)
COUNT numset;
```

DESCRIPTION **AllocateSet** allocates key buffers for multiple, simultaneous sets. Ordinarily, the c-tree Plus sequence set functions (**FirstInSet**, **NextInSet**, etc.) are only used for one set at a time. Each time **FirstInSet** or **LastInSet** is called a new set is maintained by c-tree Plus. Calling **AllocateSet** with *numset* greater than one (1) allows you to traverse as many as *numset* sets at the same time. However, to use multiple sets, you must first call **AllocateSet**, and then call **ChangeSet** before each use of the set functions: **FirstInSet**, **NextInSet**, **PreviousInSet** or **LastInSet**.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful allocation.
184	ISDP_ERR	**AllocateSet** already called. Call **FreeSet** and then **AllocateSet** if you must recall **AllocateSet**.
185	ISAL_ERR	No memory available for set buffers.

EXAMPLE

```
if (AllocateSet(5))
   printf("\nSet Allocation Error = %d\n",isam_err);
```

SEE ALSO **ChangeSet**, **FreeSet**, **FirstInSet**, **LastInSet**, **PositionSet**, **NextInSet** and **PreviousInSet**.

Begin

Mark beginning of transaction.

SHORT NAME TRANBEG

TYPE Low level data file function

DECLARATION

```
LONG Begin(trnmod);
COUNT trnmod;
```

DESCRIPTION **Begin** is used to mark the beginning of a transaction. All files that are open by this user, and that have been created with a file mode of either PREIMG or TRNLOG, will have file updates held until they are committed with a matching call to **Commit**. For a full description of transaction processing, see Chapter PRG 11.

trnmod specifies the transaction logging mode and how locks are to be used. You can specifiy either PREIMG or TRNLOG for transaction logging mode. In addition, you can "OR" in ENABLE to automatically invoke **LockISAM(ENABLE)**.

NOTE: if you use a *trnmod* of PREIMG, then trying to update a file created with TRNLOG will cause an error TTYP_ERR.

RETURN **Begin** returns a transaction marker if successful, or a zero if an error occurs. Check the value in *uerr_cod* for the error code.

Value	Symbolic Constant	Explanation
70	TEXS_ERR	transaction already pending - you cannot start one until the prior one is completed or aborted.
99	TTYPE_ERR	transaction type / *trnmod* conflict.
150	SHUT_ERR	Server is in the process of shutting down.

EXAMPLE

```
COUNT savepoint;

VOID domaster()
{Begin(ENABLE|TRNLOG);                  /*start transaction with locks*/
 while(another());                      /* get next record to add */
 { savepoint = SetSavePoint();          /* get save point at beginning
                                           of each master record */
   if (add_master()<0) Abort(); /*abort if can't add master rec.*/
   dodetail();                          /* process detail records */
 }
```

```
 if (Commit(FREE)>0) printf("\nError %d in transaction",uerr_cod);
 return;
}

VOID dodetail()
{while(moredetail());              /* get next detail record to add */
 { if (add_detail()<0)             /* add details, if possible */
   { RestoreSavePoint(savepoint) /* with error, return to savept*/
     return;
   }
 }
}
```

SEE ALSO **Abort, Commit**

ChangeSet

Change current set.

SHORT NAME CHGSET

TYPE ISAM function

DECLARATION

```
COUNT ChangeSet(setnum)
COUNT setnum;
```

DESCRIPTION **ChangeSet** is called before each use of a set function (**FirstInSet**, **NextInSet**, **PreviousInSet** or **LastInSet**) when multiple sets are in use at the same time. Before **ChangeSet** can be used, **AllocateSet** must be called to reserve the necessary key buffers. (These buffers are not a concern of your program, but they are required for multiple, simultaneous sets.) *setnum* must have a value in the range from zero to *numset* - 1, where *numset* is the argument of the **AllocateSet** function.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful set change.
186	ISNM_ERR	*setnum* out of range or **AllocateSet** has not been called.

EXAMPLE

```
COUNT i,keyno0,keyno1;
struct DATAREC datrec0,datrec1;

if (AllocateSet(2))
   printf("\nSet Allocation Error = %d\n",isam_err);
if (ChangeSet(0) || FirstInSet(keyno0,&datrec0))
   printf("\nCould not get FirstInSet. Error = %d\n",
          isam_err);
if (ChangeSet(1) || LastInSet(keyno1,&datrec1))
   printf("\nCould not get LastInSet. Error = %d\n",
          isam_err);
```

SEE ALSO **AllocateSet**, **FreeSet**, **FirstInSet**, **LastInSet**, **NextInSet** and **PreviousInSet**.

ClearTranError

Clear an error in an active transaction.

SHORT NAME TRANCLR

TYPE Low level data file function

DECLARATION

```
COUNT clearTranError();
```

DESCRIPTION If an update error occurs when you are in the middle of a Transaction, you will not be able to commit that transaction. For example, an **AddRecord** call could fail because you were trying to add a duplicate key where they are not allowed. A subsequent call to **Commit** will fail, as an error has occured.

The proper way to handle this situation is to either use **RestoreSavePoint** to go back to a savepoint, or to abandon the entire transaction with **Abort**.

There is an alternative, however. **ClearTranError** can be used to clear the error flag, allowing you to continue with the transaction. Typically you will only do this if you are working with low level update functions and you know how to recover from the error that has occurred.

We do not recommend this, however, as you are now responsible for the integrity of your data.

RETURN **ClearTranError** always returns a zero (NO_ERROR).

EXAMPLE

```
COUNT         invfil;
struct invd {
   TEXT       delflg;
   LONG       part_no;
} recbuf;

part_no = getpartno();
recbuf.delflg = '\0'; /* clear delete flag */
if (AddRecord(invfil,&recbuf)) ClearTranError();
```

SEE ALSO **Begin, Commit, Abort, RestoreSavePoint.**

CloseFile

Close index or data file.

SHORT NAME CLSFIL

TYPE Low level file function

DECLARATION

```
COUNT closeFile(filno,filmod)
COUNT filno;
COUNT filmod;
```

DESCRIPTION **CloseFile** closes the file number *filno*.

The *filmod* parameter is no longer used. It has been left in this definition solely for the sake of compatibility with older versions. If you have programs that were developed with older versions of c-tree you don't have to change your CloseFile (CLSFIL) function calls. For newer programs you can leave it out, unless you are using function prototyping and receive a warning about having too few parameters. In this case you can either use a second parameter (which will be ignored) or you can change the function prototype definition in the file *ctdecl.h*.

CloseFile must be called if the file has been updated:

- data files are updated by **NewData**, **ReleaseData** and **WriteData**;
- index files are updated by **AddKey**, **DeleteKey** and **DeleteKeyBlind**.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful data file close.
24	FCLS_ERR	Could not close file. This is an unusual error which probably results when the file's control structure is overwritten.
26	FACS_ERR	File number is not assigned to a file in use.

EXAMPLE

```
COUNT  filno;
   if (CloseFile(filno,0))
     printf("\nError closing file #%d with code %d",
         filno,uerr_cod);
```

SEE ALSO **OpenFile**, **CreatesDataFile** and **CreateIndexFile**.

CloseIFile

Incremental ISAM close.

SHORT NAME CLIFIL

TYPE ISAM function

DECLARATION

```
COUNT CloseIFile(ifilptr)
IFIL *ifilptr;
```

DESCRIPTION **CloseIFile** closes the data file and index file referenced in the IFIL structure pointed to by *ifilptr*.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful close of ISAM files.
24	FCLS_ERR	Could not close file number *isam_fil*.

EXAMPLE

```
extern IFIL myfile;

if (CloseIFile(&myfile))
   printf("\nCould not close files.");
```

SEE ALSO **InitISAM**, **CreateIFile**, **OpenIFile** and **CloseISAM**. Chapter PRG 5 discusses the contents of the IFIL structure.

CloseISAM

Close all ISAM files.

SHORT NAME CLISAM

TYPE ISAM function

DECLARATION

```
COUNT CloseISAM();
```

DESCRIPTION **CloseISAM** closes all the data files and index files opened by the previous call to **OpenISAM** or **CreateISAM** and/or by calls to **CreateIFile** and **OpenIFile**. **CloseISAM** also returns the dynamic memory allocated by c-tree Plus for control blocks, index node buffers and multiple set buffers (if allocated by **AllocateSet**). If **CloseISAM** encounters difficulty closing a file, it tries to close the remaining files. The error code returned by **CloseISAM** pertains to the first file which it could not close.

Before a second call to **CreateISAM**, **InitISAM** or **OpenISAM** can be made, **CloseISAM** must be called.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful close of all ISAM files.
24	FCLS_ERR	Could not close file number *isam_fil*.

EXAMPLE

```
if (closeISAM())
   printf("\nCould not close all files.");
```

LIMITATION **CloseISAM** closes all ISAM files including those opened or created with the incremental ISAM functions.

SEE ALSO **OpenISAM**, **OpenIFile**, **CreateISAM**, **CreateIFile**, **InitISAM** and **CloseIFile**.

CloseRFile

Incremental ISAM close for files opened by Resource IFIL open **OpenFileWith-Resource**.

SHORT NAME CLRFIL

TYPE ISAM function

DECLARATION

```
COUNT closeRFile(filno)
COUNT filno;
```

DESCRIPTION **CloseRFile** closes the data file and index file opened by a **OpenFileWith-Resource** function call.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful close of ISAM files.
24	FCLS_ERR	Could not close file number *isam_fil*.
401	RNON_ERR	Resources not enabled.
408	RNOT_ERR	IFIL structure not stored in data file resource.
438	DZRO_ERR	IFIL structure not stored in data file resource.

EXAMPLE

```
extern COUNT myfile;

if (closeRFile(myfile))
  printf("\nCould not close files.");
```

SEE ALSO **InitISAM**, **CreateIFile**, and **OpenFileWithResource**.

Commit

This function will commit a transaction.

SHORT NAME TRANEND

TYPE Low level data file function

DECLARATION

```
COUNT Commit(mode);
COUNT mode;
```

DESCRIPTION All file updates for files that support transactions will be committed, which means that they will occur at this point of time. For a full description of transaction processing, see Chapter PRG 11.

The parameter *mode* can be either FREE or KEEP. All known locks, both ISAM and low level, are released if *mode* is FREE. If the mode is KEEP then the locks are kept, and must be released by the user with a **LockISAM(FREE)** call and/or appropriate **LockData** calls.

Note that if an update error has occured, such as an **AddRecord** call failing due to a duplicate key, **Commit** will not allow you to successfully close the transaction. An PNDG_ERR error code will be returned and the transaction is aborted. This error condition can be cleared prior to the call to **Commit** with a call to **ClearTranError** (a risky thing to do) or by backing up to a savepoint set before the error occurred, with **RestoreSavePoint**.

RETURN

Value	Symbolic Constant	Explanation
37	WRITE_ERR	most likely the disk is full. Transaction is aborted.
71	TNON_ERR	there is no active transaction pending.
94	PNDG_ERR	An update error has occured during one of the previous operations which has not been cleared through **RestoreSavePoint** or **ClearTranError**.

EXAMPLE

```
Begin(ENABLE|TRNLOG);                  /*start transaction with locks*/
while(another());                      /* get next record to add */
{ if (add_master()<0) Abort(); /*abort if can't add master rec.*/
  dodetail();                          /* process detail records */
}
if (Commit(FREE)>0) printf("\nError %d in transaction",uerr_cod);
```

SEE ALSO **Begin, Abort, ClearTranError, SetSavePoint, RestoreSavePoint.**

cpybuf

Copy one buffer area to another regardless of any null characters.

TYPE Utility function

DECLARATION

```
VOID cpybuf(destination, source, length);
     TEXT *destination;
     TEXT *source;
     UINT length;
```

DESCRIPTION

This function will copy *length* bytes from *destination* to *source*. It does **not** stop when it sees a null character. You can copy strings or structures with this. We strongly suggest using this function instead of **strcpy**, for several reasons. First of all, **strcpy** continues copying until it reaches a null character, terminating the source string. If you accidentally copy a string that is larger than your destination buffer, you will overwrite adjacent data or code. This can be an extremely difficult bug to find. You can use the size of the destination buffer as the limit to **cpybuf**, thereby avoiding this problem. Secondly, many constructs used in c-tree Plus are not null-terminated strings. Keys and data records can both have null characters embedded in the middle. If you use **strcpy** you will not copy the entire data item.

EXAMPLE

```
COUNT    keyno;
TEXT     target[24],idxval[23];
LONG     recbyt;

scanf("%23s",target);
recbyt = GetGTEKey(keyno,target,idxval);
while (recbyt) {
   printf("\nIndex entry = %23s   Record # = %ld",
     idxval,recbyt);
   cpybuf(target,idxval,23);
   recbyt = GetGTKey(keyno,target,idxval);
}
```

LIMITATIONS

cpybuf does not check to see if the buffer at *destination* is large enough to hold the data being moved.

CreateDataFile

Create a data file.

SHORT NAME CREDAT

TYPE Low level data file function

DECLARATION

```
COUNT CreateDataFile(datno,filnam,datlen,xtdsiz,filmod)
COUNT  datno,filmod;
VRLEN  datlen;
TEXT  *filnam;
UCOUNT xtdsiz;
```

DESCRIPTION

CreateDataFile attempts to create a new data file with the name pointed to by *filnam* and with a record length of *datlen*. Subsequent calls to other low level data file functions use *datno* to reference this file. *datno* must be in the range 0 to *fils*-1 where *fils* is the second parameter in the initialization routine **InitCTree**. If a file with the name pointed to by *filnam* already exists, **CreateDataFile** returns an error code, and the file is not opened.

filmod determines if the file contains fixed or variable length records, whether it is shared or used exclusively, and whether it is opened permanently or virtually.

The following list shows the values that can be used for *filmod*. You can combine the values by "or"ing them together, such as:

```
(VIRTUAL | EXCLUSIVE | FIXED)
```

Some values must be used exclusively of others. For instance, you cannot use both SHARED and EXCLUSIVE. You cannot use more than one mode from any line in the listing below. In addition, you **must** use one of the modes from each of the first three groups.

```
PERMANENT          VIRTUAL
EXCLUSIVE          SHARED      READFIL
FIXED              VLENGTH
CHECKLOCK
CHECKREAD
PREIMG         TRNLOG
DUPCHANEL
WRITETHRU
SUPERFILE
DISABLERES
```

NOTE: file modes CHECKLOCK, PREIMG, TRNLOG, and DUPCHANEL are available only if you are using the FairCom Server.

If the file is created with a fixed record length, then *datlen* must be greater than 4. If the file is created with a variable record length, then *datlen* is interpreted as the minimum record length for the file and may be greater than or equal to zero.

If *xtdsiz* is greater than zero, then whenever it is necessary to extend the size of the data file, it will be extended by *xtdsiz* bytes and the directory will be updated. If *xtdsiz* is set too large, then disk space will be wasted. If *xtdsiz* is too small (but greater than zero), then the time to extend the file and force a directory update will become excessive. A value of zero for *xtdsiz* causes the file to grow one data record at a time without the directory being updated until the file is closed.

When a fixed record length data file is extended by *xtdsiz* bytes, delete flags are written into the beginning of each logical record in the newly allocated space. This may cause a noticeable delay for large values of *xtdsiz.*

NOTE: except for FIXED versus VLENGTH, and PREIMG vs TRNLOG, the file mode parameter value is not permanently stored with the file. A file may be opened as a read only (READFIL) one time, SHARED another and EXCLUSIVE yet another time.

The name referenced by *filnam* must conform to the operating system environment, and usually can include directory path names and/or disk drive identifiers. The file name should be null terminated.

SUPERFILES: Superfiles and superfile members, as discussed in Chapter PRG 8, must be handled differently than standard files. When you are creating a superfile you must "OR" in the file mode SUPERFILE. Leave *xtdsiz* and *datlen* set to 0. Remember that when you create a superfile that it will consume two consecutive file numbers, *datno* and *datno + 1*. Superfile members can be a mixture of data and index files as desired. The name of the member must be formed as follows:

```
<name of superfile>!<name of member>
```

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Data file successfully created.
17	DCRAT_ERR	Could not create data file.
19	DOPN_ERR	Data file already exists. File is not opened.
21	DREC_ERR	Record length is too small. File is not created.
22	FNUM_ERR	File number is out of range.
46	FUSE_ERR	File number is already in use.

Value	Symbolic Constant	Explanation
413	SDIR_ERR	Host superfile is not open.
414	SNST_ERR	Cannot nest superfiles.

EXAMPLE

```
COUNT  datno,reclen;
UCOUNT  xtdsiz;
TEXT filnam[15];

if (CreateDataFile(datno, filnam, reclen, xtdsiz,
   (SHARED | VIRTUAL | FIXED)))
     printf("\nError %d creating file %.14s",
         uerr_cod,filnam);
```

LIMITATIONS

While **CreateDataFile** can mark a file for shared use, it is not created in the shared mode. In order to use a data file in the shared mode, it must first be created, then closed (**CloseFile**), then opened (**OpenFile**) with a shared file mode.

SEE ALSO

InitCTree, **CreateIndexFile**, and **CreateDataFileXtd**.

CreateDataFileXtd

Extended data file creation.

SHORT NAME CREDATX

TYPE Low level data file function (extended).

DECLARATION

```
COUNT
CreateDataFileXtd(datno,filnam,datlen,xtdsiz,filmod,
    permmask,groupid,fileword)
COUNT  datno,filmod;
VRLEN  datlen;
TEXT  *filnam;
UCOUNT xtdsiz;
LONG   permmask;
TEXT  *groupid;
TEXT  *fileword;
```

DESCRIPTION

CreateDataFileXtd is a variation of **CreateDataFile** that permits the use of the FairCom Server's security system. Please read the description of **CreateDataFile** before proceeding with this description, as we will only describe the extended features here.

permmask is the **permission mask** that will be assigned to this data file. It is formed by "OR"ing the appropriate permission constants. For more information on permission masks refer to page PRG 12-3.

groupid is a pointer to a buffer that contains the **group id** that this file is to be assigned to. The group id must be valid for the user that is creating the file. If *groupid* is null, the file will be assigned to the default group for the user. See Chapter PRG 12 for more information on groups.

fileword is an optional **file password**, as explained on page PRG 12-4. If *fileword* is null then there will be no password for this file. If one is established then every user will need to use the password to be able to open the file.

RETURN

The following error codes may be seen in addition to those shown for **CreateDataFile**.

Value	Symbolic Constant	Explanation
455	SGRP_ERR	This user does not belong to the group *groupid*.

CreateIFile

Incremental ISAM create.

SHORT NAME CREIFIL

TYPE ISAM function

DECLARATION

```
COUNT CreateIFile(ifilptr)
IFIL *ifilptr;
```

DESCRIPTION **CreateIFile** creates the data file and (optional) index file referenced by the IFIL structure pointed to by *ifilptr*. When using an incremental ISAM file, all indices associated with the data file must be contained in a single, companion index. The file name pointed to by the IFIL structure (see ifilptr->pfilnam in Chapter PRG 5) SHOULD NOT include an extent: ".dat" and ".idx" are automatically appended to the file name for the data file and index file, respectively. Note that you can use the extended version of this function, **CreateIFileXtd**, to change the file extents at run time.

If you desire the files to be created in directories that are not specified until run-time, then the file name pointed to by the IFIL structure should not be finalized until run-time.

Consecutive file numbers are assigned to the data file and any (optional) indices associated with the data file. The IFIL structure contains two file numbers: a permanent file number (dfilno), and a temporary file number (tfilno). If dfilno is less than zero, then c- tree finds the first available block of consecutive file numbers which can accommodate the data file and its indices. If such a block of numbers is found, then the first number in the block is assigned to the data file and to tfilno. If dfilno is greater than or equal to zero, then dfilno is assigned to the data file and to tfilno.

See Chapter PRG 5 for a complete description of the IFIL structure.

Unless Resources are disabled for this file, **CreateIFile** automatically stores the IFIL definition in the file as a resource. This permits the use of the function **OpenFileWithResources** to open the entire IFIL set simply by file name.

RETURN Upon successful create, ifilptr->tfilno contains the file number assigned to the data file; and the associated indices are assigned consecutive index numbers starting one (1) greater than the data file number.

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful open of ISAM files.

Value	Symbolic Constant	Explanation
16	KCRAT_ERR	Could not create index file.
17	DCRAT_ERR	Could not create data file.
18	KOPN_ERR	Tried to create existing index file.
19	DOPN_ERR	Tried to create existing data file.
20	KMIN_ERR	Key length too large for node size.
21	DREC_ERR	Record length too small.
22	FNUM_ERR	File number is out of range.
45	KLEN_ERR	Key length exceeds MAXLEN parameter in *ctoptn.h*.
46	FUSE_ERR	If *ifilptr->dfilno >= 0*
47	FINT_ERR	c-tree has not been initialized. Call **InitISAM** before the first call to **CreateIFile**.
107	IDRK_ERR	Too many index files for one data file. Increase MAX_DAT_KEY parameter in *ctoptn.h*.
109	IKRS_ERR	Too many key segments. Increase MAX_KEY_SEG parameter in *ctoptn.h*.
115	ISLN_ERR	Key segments do not equal key length.

EXAMPLE

```
extern IFIL  customer;
TEXT         full_name[64];
extern TEXT *path_name,*file_name;
extern COUNT data_no;

strcpy(full_name,path_name);
strcat(full_name,file_name);
customer->pfilnam = full_name;
if (CreateIFile(&customer))
   printf("\nCould not create %s (%d,%d)\n",
     isam_err,isam_fil);
else
   data_no = customer->tfilno;
```

LIMITATIONS

The data file and index files must be numbered consecutively, and all indices must be in a single index file.

SEE ALSO

InitISAM, **OpenIFile**, **CloseISAM**, **CreateIFileXtd**, **OpenFileWithResources**, **GetIFile** and Chapter PRG 5, which describes the ISAM parameters.

CreateIFileXtd

Extended Incremental ISAM creation.

SHORT NAME CREIFILX

TYPE Extended ISAM function

DECLARATION

```
COUNT CreateIFileXtd(ifilptr,dataextn,indxextn,
                     permmask,groupid,fileword)
IFIL *ifilptr;
TEXT  *dataextn, *indxextn,*groupid,*fileword;
LONG   permmask;
```

DESCRIPTION

CreateIFileXtd is a variation of **CreateIFile** that permits the use of the FairCom Server's security system. In addition, you can use it to change the file name extensions from the defaults that are used in **CreateIFile**. Please read the description of **CreateIFile** before proceeding with this description, as we will only describe the extended features here.

permmask is the **permission mask** that will be assigned to the files created here. It is formed by "OR"ing the appropriate permission constants. For more information on permission masks refer to page PRG 12-3.

groupid is a pointer to a buffer that contains the **group id** that this file is to be assigned to. The group id must be valid for the user that is creating the file. If *groupid* is null, the file will be assigned to the default group for the user. See Chapter PRG 12 for more information on groups.

fileword is an optional **file password**, as explained on page PRG 12-4. If *fileword* is null then there will be no password for this file. If one is established then every user will need to use the password to be able to open the file.

dataextn and *indxextn* are pointers to buffers that can be used to specify optional data and index file name extensions, respectively. The extensions can be 8 byte ASCIIZ strings. If they are null pointers then the default extension will be used, ".dat" for data files and ".idx" for index files. If you want no extension then you should pass a pointer to a buffer that contains only blanks terminated by a null character. Do not set both extensions to blanks since the index and data file names must be unique.

RETURN The following error codes may be seen in addition to those shown for **CreateIFile**.

Value	Symbolic Constant	Explanation
455	SGRP_ERR	This user does not belong to the group *groupid*.

CreateIndexFile

Create an index file.

SHORT NAME CREIDX

TYPE Low level index file function

DECLARATION

```
COUNT createIndexFile(keyno,filnam,keylen,keytyp,
                      keydup,nomemb,xtdsiz,filmod)
COUNT  keyno, keylen, keytyp, keydup, nomemb, filmod;
TEXT  *filnam;
UCOUNT xtdsiz;
```

DESCRIPTION

CreateIndexFile attempts to create a new index file with the name pointed to by *filnam* and with a key length of *keylen* bytes. *keytyp* of 0 indicates alphanumeric keys (scanned left to right), and a value of 1 indicates keys scanned right to left. *keydup* of 0 indicates no duplicates allowed, and a value of 1 indicates duplicate keys are supported.

keyno must be in the range 0 to *fils*-1 where *fils* is the second parameter in the initialization routine **InitCTree**.

Subsequent calls to other low level index file functions use *keyno* to reference this file. If a file with the name pointed to by *filnam* already exists, **CreateIndexFile** returns an error code, and the file is not opened.

nomemb specifies the number of additional indices which will reside in this index file. By combining more than one index in one physical index file, an application can reduce the file descriptors used by the application. If *nomemb* is zero (0), then no additional indices will be included. Each additional index has its own key length, key type and auto-duplication parameter. In order to set these parameters for the additional indices, **CreateIndexMember** should be called for each additional index immediately after the call to **CreateIndexFile**.

Once the index file and all its members have been created, a single call to **OpenFile** will activate all members of the index file. See **OpenFile** for a more complete discussion.

Each additional index is referenced by a file number equal to *keyno* plus the member number. For example, if *keyno* is three (3) and *nomemb* is five (5), then the index created by **CreateIndexFile** will be referenced by file number three (3); the first additional index will be referenced by key number four (4), the second by five (5), and so on through key number eight (8).

If *xtdsiz* is greater than zero, then whenever it is necessary to extend the size of the index file, it will be extended by *xtdsiz* bytes and the directory will be updated. If

xtdsiz is set too large, then disk space will be wasted. If *xtdsiz* is too small (but greater than zero), then the time to extend the file and force a directory update will become excessive. A value of zero for *xtdsiz* causes the file to grow one B-Tree node at a time without the directory being updated until the file is closed.

filmod determines whether the index is shared or used exclusively, and whether it is opened permanently or virtually.

The following list shows the values that can be used for *filmod*. You can combine the values by "or"ing them together, such as:

```
(VIRTUAL    | EXCLUSIVE)
```

Some values must be used exclusively of others. For instance, you cannot use both SHARED and EXCLUSIVE. You cannot use more than one mode from any line in the listing below. In addition, you **must** use one of the modes from each of the first three groups.

```
PERMANENT          VIRTUAL
EXCLUSIVE          SHARED     READFIL
PREIMG         TRNLOG
DUPCHANEL
WRITETHRU
```

NOTE: file modes PREIMG, TRNLOG, and DUPCHANEL are available only if you are using the FairCom Server.

NOTE: Except for PREIMG and TRNLOG, the file mode parameter value is not permanently stored with the file. A file may be opened as a read only (READFIL) one time, SHARED another and EXCLUSIVE yet another time.

The name referenced by *filnam* must conform to the operating system environment, and usually can include directory path names and/or disk drive identifiers. The file name should be null terminated.

SUPERFILES: An index file can be a member of a Superfile, as discussed in Chapter PRG 8. The name of the member must be formed as follows:

```
<name of superfile>!<name of member>
```

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Index file successfully created.

Value	Symbolic Constant	Explanation
16	KCRAT_ERR	Could not create index file. Possibly no disk or directory space, no more file descriptors, or an improper file name.
18	KOPN_ERR	Index file already exists. File is not opened.
20	KMIN_ERR	Key length is too large for node size (determined by node sectors in call to **InitCTree**). There must be room for at least four key values per node.
22	FNUM_ERR	*keyno* + *nomemb* is out of range.
23	KMEM_ERR	nomemb < 0 or nomemb > 31.
45	KLEN_ERR	Key length exceeds MAXLEN parameter in *ctoptn.h*.
46	FUSE_ERR	A file number in the range *keyno* to *keyno* + *nomemb* is already in use.

EXAMPLE

```
COUNT  keyno,keylen,keytyp,keydup,nomemb;
TEXT filnam[15];

if (CreateIndexFile(keyno,filnam,keylen,keytyp,keydup,
   nomemb,8192,(SHARED | VIRTUAL)))
     printf("\nError %d creating file %.14s",uerr_cod,
         filnam);
```

LIMITATIONS

c-tree Plus does not support duplicate keys when right to left scanning is enabled (when *keytyp* equals 1). ISAM applications should use the unsigned or signed integer key segment modes (1 and 8, respectively) to automatically accommodate differences in byte ordering. This allows a *keytyp* of zero to be used, and hence duplicates may be supported.

While **CreateIndexFile** allows a file to be marked for sharing, it does not actually create the index in the shared mode. In order for an index file to be shared by multiple users, it must first be created, then closed (**CloseFile**), then opened (**OpenFile**) with a shared file mode. It is anticipated that most application programs will use a separate module to create the files at installation time, and the main module will only open the files.

Call **SetAlternateSequence** immediately after creating the index file in order to assign an alternate collating sequence to the index file.

SEE ALSO

InitCTree, **CreateDataFile**, **CreateIndexMember**, **CreateIndexFileXtd**, and page PRG 3-6 for a discussion on how automatic support of duplicate keys is handled.

CreateIndexFileXtd

Extended index file creation.

SHORT NAME CREIDXX

TYPE Extended Low level index file function

DECLARATION

```
COUNT CreateIndexFileXtd(keyno,filnam,keylen,keytyp,
                         keydup,nomemb,xtdsiz,filmod,
                         permmask,groupid,fileword)
COUNT  keyno, keylen, keytyp, keydup, nomemb, filmod;
TEXT  *filnam;
UCOUNT xtdsiz;
LONG   permmask;
TEXT  *groupid;
TEXT  *fileword;
```

DESCRIPTION

CreateIndexFileXtd is a variation of **CreateIndexFile** that permits the use of the FairCom Server's security system. Please read the description of **CreateIndexFile** before proceeding with this description, as we will only describe the extended features here.

permmask is the **permission mask** that will be assigned to this data file. It is formed by "OR"ing the appropriate permission constants. For more information on permission masks refer to page PRG 12-3.

groupid is a pointer to a buffer that contains the **group id** that this file is to be assigned to. The group id must be valid for the user that is creating the file. If *groupid* is null, the file will be assigned to the default group for the user. See Chapter PRG 12 for more information on groups.

fileword is an optional **file password**, as explained on page PRG 12-4. If *fileword* is null then there will be no password for this file. If one is established then every user will need to use the password to be able to open the file.

RETURN

The following error codes may be seen in addition to those shown for **CreateIndexFile**.

Value	Symbolic Constant	Explanation
455	SGRP_ERR	This user does not belong to the group *groupid*.

CreateIndexMember

Create an additional index file member.

SHORT NAME CREMEM

TYPE Low level index file function

DECLARATION

```
COUNT CreateIndexMember(keyno,keylen,keytyp,
                        keydup, membno)
COUNT  keyno,keylen,keytyp,keydup,membno;
```

DESCRIPTION

CreateIndexMember attempts to add an additional index to index file number *keyno* just created by **CreateIndexFile**. This additional index file member has a key length of *keylen* bytes. *keytyp* of 0 indicates alphanumeric keys (scanned left to right), and a value of 1 indicates keys scanned right to left. *keydup* of 0 indicates no duplicates allowed, and a value of 1 indicates duplicate keys are supported.

membno must be in the range from 1 to the number of members set by the previous call to **CreateIndexFile** (see the *nomemb* parameter of **CreateIndexFile**). Subsequent calls to other low level index file functions use *keyno* + *membno* to reference this file. For example, if *keyno* is three (3) and *membno* is two (2), then this additional index file member is referenced as file number five (5).

Call **SetAlternateSequence** immediately after creating the index member in order to assign an alternate collating sequence to the index member.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Index file member successfully added.
20	KMIN_ERR	Key length is too large for node size (determined by node sectors in call to **InitCTree**). There must be room for at least four key values per node.
23	KMEM_ERR	*membno* < *1* or *membno* greater than number of additional members established for the index keyno.
45	KLEN_ERR	Key length exceeds MAXLEN parameter in *ctoptn.h*.
46	FUSE_ERR	File number *keyno* + *nomemb* is already in use.
48	FMOD_ERR	*keyno* is not assigned to an index file.

EXAMPLE

```
COUNT        keyno,keylen,keytyp,keydup,membno;
```

```
if (CreateIndexMember(keyno,keylen,keytyp,
                      keydup,membno))
   printf("\nError %d when adding index #%d to file %d",
     uerr_cod,membno,keyno);
if (AddKey(keyno + membno,"FUNNY KEY",100L,REGADD))
   printf("\nCould'nt add info to member %d. Error %d",
     membno,uerr_cod);
```

LIMITATIONS

c-tree Plus does not support duplicate keys when right to left scanning is enabled (when *keytyp* equals 1). See the Limitations discussion under **CreateIndexFile**.

SEE ALSO

SetAlternateSequence, **InitCTree**, **CreateIndexFile**, **OpenFile** and page PRG 3-6 for a discussion on how automatic support of duplicate keys is handled.

CreateISAM

Create all the ISAM files.

SHORT NAME CREISAM

TYPE ISAM function

DECLARATION

```
COUNT CreateISAM(filnam)
TEXT *filnam;
```

DESCRIPTION

CreateISAM opens the parameter file whose name is pointed to by *filnam* and creates the data and index files specified in the parameter file. Application programs using the ISAM level functions must make a call to **CreateISAM**, **InitISAM** or **OpenISAM** before any other c-tree Plus functions are used. If any one of these three functions has been called, then **CloseISAM** must be called before anyone of them can be invoked again. In part, this implies that no more than one ISAM parameter file may be active at one time. However, the incremental ISAM functions allow for more flexibility in file manipulations.

In order to use incremental ISAM opens and creates along with a parameter file, you must increase the value of the second parameter in the parameter file (idxs) by the number of data files and indices to be utilized with the incremental routines. For example, if the incremental routines would be opening two additional data files, each with five indices, then the idx parameter in your ISAM parameter file should be increased by 12.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful open of ISAM files.
10	SPAC_ERR	Not enough space for file structures and buffers.
11	SPRM_ERR	Illegal **InitCTree** parameters in first record of parameter file.
16	KCRAT_ERR	Could not create index file.
17	DCRAT_ERR	Could not create data file.
18	KOPN_ERR	Tried to create existing index file.
19	DOPN_ERR	Tried to create existing data file.
20	KMIN_ERR	Key length too large for node size.
21	DREC_ERR	Record length too small.

Value	Symbolic Constant	Explanation
22	FNUM_ERR	File number is out of range.
45	KLEN_ERR	Key length exceeds MAXLEN parameter in *ctoptn.h* .
102	INOD_ERR	Could not open ISAM parameter file.
103	IGIN_ERR	Could not read first record of ISAM parameter file.
104	IFIL_ERR	Too many files specified. Increase MAXFIL in *ctoptn.h*
106	IDRI_ERR	Could not read data file description record in parameter file.
107	IDRK_ERR	Too many index files for one data file. Increase MAX_DAT_KEY parameter in *ctoptn.h*.
108	IMKY_ERR	*keyno* for index member is out of sequence. *keyno* must equal the host index keyno plus the member number.
109	IKRS_ERR	Too many key segments. Increase MAX_KEY_SEG parameter in *ctoptn.h*.
110	ISRC_ERR	Could not read key segment description record in parameter file.
111	IKRI_ERR	Could not read index file description record in parameter file.
115	ISLN_ERR	Key segments do not equal key length.
117	IMRI_ERR	Could not read index member record.
125	IINT_ERR	c-tree Plus already initialized. Call **CloseISAM** before recalling **CreateISAM**.
182	IKSR_ERR	No room for r-tree temporary files. Increase MAXFIL in *ctoptn.h*.

EXAMPLE

```
TEXT        isam_par[64];

printf("\nEnter ISAM parameter file name: ");
scanf("%63s",isam_par);
if (CreateISAM(isam_par))
   printf("\n%s files could not be created. (%d  %d)",
     isam_err,isam_fil);
```

LIMITATIONS

It is not possible to open some of the files specified in the parameter file and create others. They must all be created by **CreateISAM**. However, the incremental ISAM functions permit on the fly opening and closing of ISAM files.If you are adding a new index file to an established set of ISAM files, you should :

- update the parameter file to include the new index;
- use the rebuild utility to create and build the new index file (respond no when asked to force rebuild); and
- then use **OpenISAM** to actually use the files.

SEE ALSO

CloseISAM, **InitISAM**, **OpenISAM**, **CreateIFile**, **OpenIFile**, **CreateISAMXtd**, and Chapter PRG 5 which describes the parameter file contents.

CreateISAMXtd

Extended ISAM file creation routine.

SHORT NAME CREISAMX

TYPE Extended ISAM function

DECLARATION

```
COUNT CreateISAMXtd(filnam,userprof,userid,userword,
                    servname,permmask,groupid,fileword)
TEXT *filnam;
COUNT userprof;
TEXT *userid;
TEXT *userword;
TEXT *servname;
LONG   permmask;
TEXT  *groupid;
TEXT  *fileword;
```

CreateISAMXtd is a variation of **CreateISAM** that permits the use of the FairCom Server's security system, as well as setting the user profile mask. Please read the description of **CreateISAM** before proceeding with this description, as we will only describe the extended features here.

userprof is the **user profile mask**. If you set this to the value USERPRF_NTKEY you will **disable** the **automatic TransformKey** feature. See the function description for **TransformKey** for more information. If you do not wish to disable this feature, set *userprof* to zero.

userid is a pointer to a buffer containing your user ID. See page PRG 12-2 for more information on user ID's.

userword is a pointer to a buffer containing your user password. See page PRG 12-2 for more information on user passwords.

servname is a pointer to a server name if you are going to use a server other than the default server, in a multi-server environment. See the Server documentation for more information.

permmask is the **permission mask** that will be assigned to the files created here. It is formed by "OR"ing the appropriate permission constants. For more information on permission masks refer to page PRG 12-3.

groupid is a pointer to a buffer that contains the **group id** that this file is to be assigned to. The group id must be valid for the user that is creating the file. If *groupid* is null, the file will be assigned to the default group for the user. See Chapter PRG 12 for more information on groups.

fileword is an optional **file password**, as explained on page PRG 12-4. If *fileword* is null then there will be no password for this file. If one is established then every user will need to use the password to be able to open the file.

All files in the parameter file are assigned to the same *permmask*, *groupid*, and *fileword*.

RETURN

The following error codes may be seen in addition to those shown for **CreateISAM**.

Value	Symbolic Constant	Explanation
450	LUID_ERR	Invalid user ID.
451	LPWD_ERR	Invalid user password.
453	NSRV_ERR	Invalid server name.
455	SGRP_ERR	This user does not belong to the group *groupid*.

CurrentFileOffset

Returns the byte position (file address or offset) of the current ISAM record for the specified file.

SHORT NAME GETCURP

TYPE ISAM function

DECLARATION

```
LONG CurrentFileOffset(datno);
COUNT datno;
```

DESCRIPTION **CurrentFileOffset** returns the byte position (file address) of the current ISAM record for file *datno*. Use this if you need to access the current ISAM record with a low level function.

RETURN Returns the byte position of the current ISAM record. Zero is returned if there is no current ISAM record for the specified *datno*, or if an error occurs. *isam_err* will be set as follows:

Value	Symbolic Constant	Explanation
22	FNUM_ERR	*datno* out of range.
26	FACS_ERR	*datno* is not active.

EXAMPLE

```
LONG curptr;
COUNT datfil;
TEXT  recbuf[320];

if (FirstRecord(datfil,recbuf) == NO_ERROR)
   printf("\nFirst Record is at %ld",
          CurrentFileOffset(datfil));
```

SEE ALSO **CurrentISAMKey**

CurrentISAMKey

Create an ISAM key from the current ISAM record.

SHORT NAME GETCURK

TYPE ISAM function

DECLARATION

```
VOID *CurrentISAMKey(keyno, idxval, plen);
COUNT keyno;
VOID *idxval;
VRLEN *plen;
```

DESCRIPTION **CurrentISAMKey** looks at the current ISAM key buffer for index file *keyno* and creates a well formed ISAM key. This key is placed in *idxval*. The length of the key is placed in **plen* if *plen* is not NULL.

RETURN **CurrentISAMKey** returns a pointer to the key value. A NULL signifies an error.

Value	Symbolic Constant	Explanation
22	FNUM_ERR	*keyno* out of range.
26	FACS_ERR	*keyno* is not active.
48	FMOD_ERR	*keyno* does not reference an index file.

EXAMPLE

```
COUNT keyfil;
TEXT  keybuf[50];
VRLEN keylen:

if (CurrentISAMKey(keyfil,&keybuf,&keylen) == NULL)
   printf("\nError %d with file %d",isam_err,isam_fil);
```

LIMITATIONS No check is made to determine if *idxval* points to a region sufficiently large to accept the key. If the area is too small, either code or data will be clobbered. You can use **GetFileInfo**(keyno,keylen) to obtain the key length (as a long integer).

DeleteFile

Delete c-tree Plus file.

SHORT NAME DELFIL

TYPE Low level

DECLARATION

```
COUNT DeleteFile(filno)
COUNT filno;
```

DESCRIPTION **DeleteFile** deletes the file opened with file number *filno* if and only if the file has been opened in the EXCLUSIVE mode. The file is closed (by c-tree Plus) before it is deleted.

NOTE: any c-tree Plus file may be deleted with **DeleteFile**, whether it has been opened or created with a low level function, an ISAM parameter file, or with incremental ISAM structures; provided the file has been assigned an EXCLUSIVE file mode.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful file deletion.
55	DLTF_ERR	File could not be deleted. Does any process have the file opened?
56	DLTP_ERR	File is not opened in the EXCLUSIVE mode. Delete is not performed.

EXAMPLE

```
COUNT       tempno;

if (CreateDataFile(tempno =
5,"temp.dat",1004,0,EXCLUSIVE)) {
   printf("\nCould not create temporary file (%d)\n",
     uerr_cod);
temporary_processing(tempno);
if (DeleteFile(tempno)) {
   printf("\nCould not delete temporary file (%d)\n",
     uerr_cod);
```

LIMITATIONS **DeleteFile** first closes the file before deletion. This permits another process to have a small window of opportunity to gain control of the file in which case the delete will fail and result in an error DLTF_ERR (55).

DeleteKey

Delete key value verifying associated data record position.

SHORT NAME DELCHK

TYPE Low level index file function

DECLARATION

```
COUNT DeleteKey(keyno,target,recbyt)
COUNT   keyno;
VOID   *target;
LONG    recbyt;
```

DESCRIPTION **DeleteKey** deletes the key value pointed to by *target* from index file *keyno* if such a key value exists and the associated data record position in the index matches *recbyt*.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful deletion.
3	KMAT_ERR	Key found
4	KDEL_ERR	Cannot find key value to delete.

EXAMPLE

```
LONG          recbyt;
COUNT         keyno;
TEXT          target[24];

if (DeleteKey(keyno,target,recbyt))
   printf("\nUnsuccessful deletion (code =
%d).",uerr_cod);
```

LIMITATIONS In order to support simultaneous, multi-user updates with guaranteed deadlock free node locks, c-tree Plus does not merge "underflowed" nodes (i.e., nodes which are less than half-full). The B-Tree stays balanced in the essential sense that each key value is the same distance from the root as any other key value, but there is no guarantee that each node is at least half full. However, the deleted space can be re-used by new key values.

This limitation is most serious when the key values are systematically deleted from one end of the key value range and systematically added to the other end of the range. Over time, this will lead to a significant amount of wasted disk space.

If an application deletes a substantial portion of the index entries, disk space can be saved by compacting the index via the CTCMPC utility program.

SEE ALSO

DeleteKeyBlind (which deletes key values when the associated data record position is unknown).

DeleteKeyBlind

Delete key value when associated data record position is not known.

SHORT NAME DELBLD

TYPE Low level index file function

DECLARATION

```
LONG DeleteKeyBlind(keyno,target)
COUNT keyno;
VOID *target;
```

DESCRIPTION **DeleteKeyBlind** deletes the key value pointed to by target from index file *keyno,* if such a key value exists. **DeleteKeyBlind** cannot be used when the index file supports duplicate keys.

RETURN If the key value is found in the index, it is deleted and **DeleteKeyBlind** returns the associated data record position. If it is not found, or some error is detected, **DeleteKeyBlind** returns a zero (which is never a legal data record position), and uerr_cod is set as follows:

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful deletion.
4	KDEL_ERR	Cannot find key value to delete.
5	KBLD_ERR	Cannot call **DeleteKeyBlind** when duplicate keys are supported.

EXAMPLE

```
LONG        recbyt;
COUNT       keyno;
TEXT        target[24];

keyno = 1;
printf("\nEnter key value: “);
scanf("%23s",target);
if (recbyt = DeleteKeyBlind(keyno,target))
   printf("\n%s deleted (data record position = %ld).",
     target,recbyt);
else
   printf("\nUnsuccessful deletion (code =
%d).",uerr_cod);
```

LIMITATIONS

Cannot be used with an index which accepts duplicate key values since without knowledge of the associated data record position, there is no way to distinguish among identical key values.

SEE ALSO

DeleteKey which deletes key values provided you already know the associated data record position.

DeleteRecord

Delete current fixed length ISAM record.

SHORT NAME DELREC

TYPE ISAM fixed length record function

DECLARATION

```
COUNT DeleteRecord(datno)
COUNT datno;
```

DESCRIPTION **DeleteRecord** deletes the current fixed length ISAM record for data file *datno*. If successful, this causes all associated index files to be updated and returns the data record to the pool of available records for data file *datno*. After the deletion, the current ISAM record information stays unchanged! That is, even though the current record has just been deleted, the information in the current ISAM buffer is not disturbed so that you can scan in either direction through the data via the **NextRecord** or **PreviousRecord** routines.

In multi-user systems, you should be sure to invoke **LockISAM(ENABLE)** before the ISAM routine which reads the record to be deleted. **LockISAM(FREE)** should be called after **DeleteRecord**.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful ISAM deletion. Current ISAM record remains unchanged.
4	KDEL_ERR	Could not delete key value.
48	FMOD_ERR	*datno* is assigned to a variable length data file.
57	DADV_ERR	Proper lock not found by the FairCom Server. See *ctsrvr.c*.
100	ICUR_ERR	No current ISAM record for data file.

EXAMPLE

```
COUNT       invfil,part_no_idx;
LONG        part_key;
struct invd {
   TEXT     delflg;
   LONG     part_no;
   TEXT     part_name[60];
} recbuf;
```

```
printf("\nEnter part number to delete: ");
scanf("%ld",&part_key);
if (LockISAM(ENABLE) || GetKey(part_no_idx,&part_key,
   &recbuf) || DeleteRecord(invfil))
     printf("\nCould not delete PART #%ld (%d %d).",
         part_key,isam_err,isam_fil);
LockISAM(FREE);
```

LIMITATIONS

DeleteRecord causes the first byte of the data record to be set to ff (hex) as a delete flag. This flag is tested by all ISAM functions which read a data record. If the first byte is equal to ff (hex), error 114 (IRED_ERR) is returned. The delete flag is also used by the rebuild utility to determine if a record is deleted. When you define your data record structure, the most straightforward approach is to reserve the first byte of the data record for the delete flag. You may store data in the first byte as long as the data will never be equal to ff (hex).

DeleteResource

Delete Resource from file.

SHORT NAME DELRES

TYPE Resource function

DECLARATION

```
COUNT DeleteResource(filno, resptr);
COUNT filno;
VOID  *resptr;
```

DESCRIPTION **DeleteResource** is used to delete a Resource from data file *filno*. If there is a record lock on this Resource the lock is automatically freed.

resptr points to a Resource Data Block as shown below. This is the same structure as used in the other Resource functions.

Byte Offset	Data Type	
0 - 3	unsigned long integer	Resource Type
4 - 7	unsigned long integer	Resource Number
8 - m	null terminated character string	Resource Name

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful Resource deletion.
57	DADV_ERR	Proper lock not found by the FairCom Server.
401	RNON_ERR	Resources have not been enabled for this file.
408	RNOT_ERR	Resource not found.

EXAMPLE

```
#include "ctstdr.h"
#include "ctdecl.h"

structure {                              /* Resource Data Block */
   ULONG  resource_type;
   ULONG  resource_number;
   TEXT   resource_name[80];
} my_resource;
```

```
my_resource.resource_type    = 0x10001L;    /* 65537 */
my_resource.resource_number  = 100;

if (DeleteResource(datno,&my_resource,) == 0 )
      printf("\nThe resource has been deleted");
else
      printf("\nCould not delete resource. Error #%d",
            uerr_cod);
```

SEE ALSO **AddResource, UpdateResource, GetResource.**

DeleteVRecord

Delete current variable length ISAM record.

SHORT NAME DELVREC

TYPE ISAM variable length record function

DECLARATION

```
COUNT DeleteVRecord(datno)
COUNT datno;
```

DESCRIPTION **DeleteVRecord** deletes the current variable length ISAM record for data file *datno*. If successful, this causes all associated index files to be updated and returns the data record to the pool of available records for data file *datno*. After the deletion, the current ISAM record information stays unchanged! That is, even though the current record has just been deleted, the information in the current ISAM buffer is undisturbed so that you can scan in either direction through the data via the **NextRecord** or **PreviousRecord** routines.

In multi-user systems, you should be sure to invoke **LockISAM(ENABLE)** before the ISAM routine which reads the record to be deleted. **LockISAM(FREE)** should be called after **DeleteVRecord**.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful ISAM deletion. Current ISAM record remains unchanged.
4	KDEL_ERR	Could not delete key value. Has a previous **ReWrite-Record** updated a key value AND not used separate buffers for the current ISAM record and the updated record?
48	FMOD_ERR	datno is assigned to a fixed length data file.
57	DADV_ERR	Proper lock not found by the FairCom Server. See *ctsrvr.c*
100	ICUR_ERR	No current ISAM record for data file.

EXAMPLE

```
COUNT          invfil,part_no_idx;
LONG           part_key;
struct invd    {
   LONG        part_no;
```

```
    TEXT        part_name[120];
} recbuf;

printf("\nEnter part number to delete: ");
scanf("%ld",&part_key);
if (LockISAM(ENABLE) ||
GetRecord(part_no_idx,&part_key,
    &recbuf) || DeleteVRecord(invfil))
      printf("\nCould not delete PART #%ld (%d %d).",
          part_key,isam_err,isam_fil);
LockISAM(FREE);
```

DoBatch

Perform operations on a group of records.

SHORT NAME BATSET

TYPE ISAM function.

DECLARATION

```
COUNT DoBatch(filno, request, bufptr, bufsiz, mode)
COUNT filno;
VOID *request;
VOID *bufptr;
VRLEN bufsiz;
UCOUNT mode;
```

DESCRIPTION

DoBatch is an ISAM level function that will attempt to perform a specified function on a group of records that have keys that match a partial key value. *filno* specifies the index file to be used to retrieve the related records. *bufptr* is a pointer to a region into which the group of related records will be retrieved, if specified. *mode* is a value that specifies the type of action to be taken. *request* is a pointer to a *Partial Key Request* (or PKEYREQ) structure, which defines the set of records to be acted on.

You will make at least two calls to **DoBatch**. The first call specifies the type of action to be taken. The valid *mode* parameters are **BAT_GET** and **BAT_DEL**. You can "or" in several values to modify the effect of the call, as shown in the table below.

If you are processing more records than can be fit into the buffer pointed to by *bufptr*, you can make additional calls to **DoBatch** to retrieve the rest. You will use a *mode* value of **BAT_NXT**.

When you are finished with a batch (even if you have not processed all of the records through additional **DoBatch** calls) you should call **DoBatch** one more time, with a *mode* value of **BAT_CAN**. This closes the batch, frees all locks, and allows you to start another batch. In some cases when you make this last call you may get a BTNO_ERR, signifying that there is no batch active. Ignore this error in this situation. There are times where the **DoBatch** function may close the batch automatically. However, if you don't close the batch when you are done you may get a BTIP_ERR or BTST_ERR error code when attempting to open your next batch.

The values that can be used for the *mode* parameter are: (*see next page*)

Symbolic Constant	Explanation
BAT_GET	Retrieve the records that have keys that match the value in the PKEYREQ structure.
BAT_DEL	Delete the records that have keys that match the value in the PKEYREQ structure.
BAT_VERIFY	Verify that the keys in the index match the values in the key fields of the record.
BAT_RET_REC	Place the data records in the buffer pointed to by *bufptr*.
BAT_RET_POS	Place only the record positions in the buffer pointed to by *bufptr*.
BAT_LOK_RED	Place a **read** lock on each record that matches the Partial Key Request.
BAT_LOK_WRT	Place a **write** lock on each record that matches the Partial Key Request.
BAT_COMPLETE	**DoBatch** will return a successful completion value only if ALL matching records are successfully locked. You must specify either BAT_LOK_RED or BAT_LOK_WRT in addition to this.
BAT_NXT	Use this in a second and subsequent call to **DoBatch** to process the next group of matching records if they all couldn't fit in the buffer on the first call.
BAT_CAN	Use this in a final call to cancel processing of the remaining records of a batch and free all locks.

request points to a *Partial Key Request* (PKEYREQ) structure. This is defined as:

```
typedef struct pkeyreq {
    LONG  btotal;          /* total entries in set      */
    LONG  bavail;          /* # of entries locked       */
    LONG  breturn;         /* # of objects in buffer    */
    COUNT siglen;          /* significant length        */
    TEXT  target[MAXLEN];/* partial key target         */
    }  PKEYREQ;
```

When you call **DoBatch** you will pass a pointer to a PKEYREQ structure. The only fields that must be initialized are *target*, which will contain the portion of a properly formed key that you want to match with, and *siglen*, which specifies the number of bytes of the key that is to be considered significant. All keys in the appropriate index that match the first *siglen* bytes of *target* will be processed.

After the **DoBatch** function has been called you can check the values in the PKEYREQ structure for the results. *btotal* will tell you how many records match the partial key target. If you ask for locks on the records, *bavail* will tell you how many of the records were locked successfully. *breturn* specifies the number of records or record positions that were passed back in the buffer region.

NOTE: if neither BAT_RET_REC nor BAT_RET_POS are specified, **DoBatch** will not return any values to the buffer at *bufptr*. The values in the PKEYREQ structure will still be updated.

If BAT_RET_POS is specified, then the buffer region contains an array of 4 byte integerss, each specifying the byte position of a record in the partial key set. If BAT_RET_REC is specified, then the buffer region contains a series of entries comprised of the record position (4 bytes) followed by the actual record. If the file contains variable length records then the 4 byte record position is followed by a 4 byte record length and then the variable length record.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful processing of the batch call. If BAT_COMPLETE was specified, all of the records were locked successfully.
42	DLOK_ERR	If BAT_COMPLETE was specified
101	INOT_ERR	No match for target key value found.
421	BTFL_ERR	Filter not supported yet. You have tried to access a feature that has not been implemented yet. The most likely cause would be an invalid value in the *mode* parameter.
422	BTFN_ERR	Other functions not supported yet. You have tried to access a feature that has not been implemented yet. The most likely cause would be an invalid value in the *mode* parameter.
423	BTIC_ERR	Batch incomplete. You specified BAT_COMP in the *mode* parameter
424	BTAD_ERR	Add list error. This is an internal processing error - possibly you have a memory overwrite problem of some sort.

Value	Symbolic Constant	Explanation
425	BTIP_ERR	Batch already in progress. You are trying to start a new batch before you have completed the previous one. You must complete the previous batch with a **DoBatch** call with a mode of BAT_CAN before you can proceed with a new batch.
426	BTNO_ERR	No batch active.
427	BTST_ERR	Status info already returned. You have tried two consecutive **DoBatch** calls requesting status info (you did not specify either BAT_RET_REC or BAT_RET_POS) without an intervening call with BAT_CAN.
428	BTMT_ERR	No more info. You made a call with a *mode* of BAT_NXT, but all of the records have been accessed.
429	BTBZ_ERR	*bufsiz* is too small for a record from this file.
430	BTRQ_ERR	Request is empty. NULL *request* pointer

EXAMPLE

```
COUNT    keyfil;
PKEYREQ  batsamp;   /*Partial Key Request structure*/
TEXT     recbuf(1024); /* buffer for records */

batsamp.siglen = 4;  /* first 4 bytes significant*/
sprintf(batsamp.target,"DISK");/*partial key value*/
if DoBatch(keyfil,&batsamp,recbuf,1024,
    BAT_GET|BAT_RET_REC|BAT_COMPLETE|BAT_LOK_WRT)
    { printf("\nBatch error %d",isam_err);
      return;
    }
printf("\n%ld entries match",batsamp.btotal);
process_batch(); /* process the batch */
DoBatch(keyfil,NULL,NULL,0L,BAT_CAN); /*close the
batch */
```

SEE ALSO

Please refer to Chapter PRG 8 for a detailed explanation of this function.

EnableResource

Enable Resources in a file that was created without resources being enabled.

SHORT NAME

ENARES

TYPE

Resource function

DECLARATION

```
COUNT EnableResource(filno);
COUNT filno;
```

DESCRIPTION

When you create a file with **CreateISAM**, **CreateIFile**, **CreateIndexFile** or **CreateDataFile**, you can disable resources for that file. If, at some later date, you decide that you want to use resources with that file you can use **EnableResource** to turn that feature on.

The file must already be open, and it must be opened with a file lock mode of EXCLUSIVE. Otherwise you will receive an error. *filno* is the file number for the file that you are working with.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful Resource enable.
400	RCRE_ERR	Resources have already been enabled for this file.
402	RXCL_ERR	*datno* has not been opend with an EXCLUSIVE lock.

EXAMPLE

```
COUNT datafil;

if (OpenFile(datafil,"test.dat",EXCLUSIVE))
    printf("\nCannot open file, error %d",uerr_cod):
else
{   if EnableResource(datafil))
        printf("\nCannot enable resource, err %d",
                uerr_cod);
    CloseFile(datafil,0);
}
```

SEE ALSO

CreateDataFile, **CreateISAM**, **CreateIFile**, **CreateIndexFile**.

EstimateKeySpan

Find the approximate number of entries between two key values.

SHORT NAME ESTKEY

TYPE Low level index file function

DECLARATION

```
LONG EstimateKeySpan(keyno,target1,target2)
COUNT keyno;
VOID *target1,*target2;
```

DESCRIPTION **EstimateKeySpan** finds the approximate number of entries between the two key values pointed to by *target1* and *target2*, respectively. **5EstimateKeySpan** does not traverse the index to compute the values; instead it makes about ten calls to **KeyAtPercentile** to determine the relative location of the target values. **Estimate-KeySpan** does not require *target1* and *target2* to be in ascending order.

EstimateKeySpan can be used to select among alternate search paths when a retrieval request may be satisfied by more than one index.

RETURN If **EstimateKeySpan** is successful, it returns the approximate number of key value entries located between *target1* and *target2*. If an error occurs or the index is empty, then **EstimateKeySpan** returns a zero. After a call to **EstimateKeySpan** which returns a zero, check the value of uerr_cod: if uerr_cod is non-zero, an error condition was detected; otherwise, the index is empty.

EXAMPLE

```
LONG      EstimateKeySpan();
LONG      retval;
COUNT         keyno;
TEXT          t1[24],t2[24];
scanf("%23s %23s",t1,t2);
if (retval = EstimateKeySpan(keyno,t1,t2))
   printf("\n %ld entries between %.23s and %.23s",
     retval,t1,t2);
else if (uerr_cod)
   printf("\n error %d occurred",uerr_cod);
else
   printf("\n the index is empty");
```

LIMITATIONS The number of index entries must be maintained by the index header record (which is the default when c-tree Plus is shipped).

EstimateKeySpan should not be used in situations which require precise results since it is based on an approximate method.

SEE ALSO **KeyAtPercentile**

FirstInSet

Read the first data record in the set defined by target.

SHORT NAME FRSSET

TYPE ISAM function

DECLARATION

```
COUNT FirstInSet(keyno,target,recptr,siglen)
COUNT keyno;
VOID *target,*recptr;
COUNT siglen;
```

DESCRIPTION

FirstInSet reads the first data record in the set of data records whose keys match the target key in the first *siglen* bytes. If successful, this record becomes the current ISAM record for the associated data file. A successful **FirstInSet** defines a current key value set, and subsequent calls to **NextInSet** and **PreviousInSet** will read the other records in the set (i.e., those records whose keys also match the first *siglen* bytes of *target*). If an error occurs or no key value matches the target, the current ISAM record is not updated, and no key value set is defined.

If the associated data file has variable length records, then only the minimum record length portion (defined by *datlen* in the original call to **CreateDataFile**) is actually read into the buffer pointed to by *recptr*. If you wish to read the entire variable length record into the same or a different buffer, issue a call to **ReReadVRecord** after the call to **FirstInSet**. Note that **ReReadVRecord** requires you to specify the size of the buffer area so that it can check if sufficient space is available.

FirstInSet can be used to perform an equality search for duplicate keys. Set the target to the key value of interest, and set the *siglen* parameter to the key length less the four bytes of the suffix.

Notice that you do not directly identify the data file number involved. The ISAM parameter file described in Chapter PRG 5 contains the correspondence between the index file number (*keyno*) and the associated data file.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful update of current ISAM record and definition of key value set.
33	DNUL_ERR	*recptr* is NULL. No data file read performed.

Value	Symbolic Constant	Explanation
42	DLOK_ERR	Could not get lock on data record. No data file read performed.
101	INOT_ERR	No matching key value found.
119	SKTY_ERR	*keyno* must reference a left-to-right scan key type.

EXAMPLE

```
COUNT          keyfil;
TEXT           target[24],recbuf[320];

printf("\nEnter set value:");
scanf("%23s",target);
FirstInSet(keyfil,target,recbuf,strlen(target));
while (isam_err == NO_ERROR) {
   process_data();
   NextInSet(keyfil,recbuf);
}
```

LIMITATIONS

FirstInSet can only be called for keys which are scanned from left to right. *keytyp*'s of 1, 2 and 3 are not permitted.

No check is made to determine if *recptr* points to a region sufficiently large to accept a data record. If the area is too small, either code or data will be clobbered.

SEE ALSO

NextInSet, **PreviousInSet**, **LastInSet**, **PositionSet** and **ChangeSet**.

FirstKey

Find the first entry in an index file.

TYPE Low level index file function

DECLARATION

```
LONG FirstKey(keyno,idxval);
COUNT keyno;
VOID *idxval;
```

DESCRIPTION **FirstKey** searches index file *keyno* for the first entry in the index. If successful, it copies the first index entry into the space pointed to by *idxval.*

RETURN **FirstKey** returns the data record position associated with the first entry in the index file. If the index is empty or an error is detected, then **FirstKey** returns a zero. If **FirstKey** returns a zero, then check uerr_cod to see if an error occurred: if uerr_cod is also zero, then the index is empty; otherwise an error condition was detected.

EXAMPLE

```
TEXT          idxval[10];
COUNT         keyno;

if (FirstKey(keyno,idxval))
   printf("\nFirst index entry = %.10s");
else {
   printf("\nEither index is empty or an error
occurred.");
   if (uerr_cod)
     printf("\Error = %d",uerr_cod);
   else
     printf("\Index is empty.);
}
```

LIMITATIONS No check is made to determine if idxval points to a region sufficiently large to accept a key value from the index file. If the area is too small, either code or data will be clobbered.

SEE ALSO **LastKey**.

FirstRecord

Read the first data record.

SHORT NAME FRSREC

TYPE ISAM function

DECLARATION

```
COUNT FirstRecord(filno,recptr)
COUNT filno;
VOID *recptr;
```

DESCRIPTION

If *filno* designates an index file, then **FirstRecord** reads the first data record based on the key sequential order of entries in index file number *filno*. If *filno* designates a data file, **FirstRecord** reads the first active data record in physical sequential order. If successful, the first record becomes the current ISAM record for the associated data file. If an error occurs or there are no entries, the current ISAM record is not updated.

If *filno* designates a data file, and this file is a member of a Superfile, then FirstRecord may not reliably return the first physical active data record.

If the data file has variable length records, then only the minimum record length portion (defined by *datlen* in the original call to **CreateDataFile**) is actually read into the buffer pointed to by *recptr*. If you wish to read the entire variable length record into the same or a different buffer, issue a call to **ReReadVRecord** after the call to **FirstRecord**. Note that **ReReadVRecord** requires you to specify the size of the buffer area so that it can check if sufficient space is available.

Notice that if **FirstRecord** is called with an index number, you do not directly identify the data file number involved. The ISAM parameters described in Chapter PRG 5 contain the correspondence between the index number *(filno)* and the associated data file.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful update of current ISAM record.
33	DNUL_ERR	*recptr* is NULL. No data file read performed.
42	DLOK_ERR	Could not get lock on data record. No data file read performed.
101	INOT_ERR	No active entries.

EXAMPLE

```
COUNT          keyfil,datfil;
TEXT           recbuf[320];

if (FirstRecord(keyfil,recbuf) == NO_ERROR &&
   ReReadVRecord(datfil,recbuf,320) == NO_ERROR)
     printf("\nSuccessful FirstRecord.");
```

LIMITATIONS

No check is made to determine if *recptr* points to a region sufficiently large to accept a data record. If the area is too small, either code or data will be clobbered.

SEE ALSO

NextRecord, **PreviousRecord** and **LastRecord**.

FreeSet

Free space allocated for multiple sets.

SHORT NAME FRESET

TYPE ISAM function

DECLARATION

```
COUNT FreeSet()
```

DESCRIPTION **FreeSet** should be called when it is desired to free the space allocated by **AllocateSet** for multiple, simultaneous sets. Ordinarily, the only time it is necessary to call **FreeSet** is when you desire to increase the number of simultaneous sets. In this case, you call **FreeSet** to free the buffers and then **AllocateSet** with the new upper limit on sets. **CloseISAM** will free multiple set buffers if they have not already been freed via **FreeSet**.

RETURN **FreeSet** should always return a zero (NO_ERROR) value. (Even if **FreeSet** is called when no buffers are allocated, it will simply return a zero without attempting to free any space.)

EXAMPLE

```
if (AllocateSet(5) == 0)
   printf("\nRoom for 5 sets\n");
FreeSet();
if (AllocateSet(10) == 0)
   printf("\nRoom for 10 sets\n");
```

SEE ALSO **AllocateSet**.

GetAlternateSequence

Retrieve information from a data file relating to an alternate collating sequencethat has been stored as a Resource.

SHORT NAME GETALTTSEQ

TYPE Resource function

DECLARATION

```
COUNT GetAlternateSequence(keyno,altseq)
COUNT keyno, *altseq;
```

DESCRIPTION **GetAlternateSequence** returns the alternative collating sequence into the array of 256 short integers pointed to by *altseq*.

RETURN

Value	Symbolic Constant	Explanation
22	FNUM_ERR	*keyno* out of range.
26	FACS_ERR	*keyno* is not active.
401	RNON_ERR	resources not enabled.
408	RNOT_ERR	no alternative collating sequences assigned to index file.
438	DZRO_ERR	no alternative collating sequence assigned to index file.

SEE ALSO **SetAlternateSequence**

GetDODA

Retrieve information from a record schema stored as a resource.

SHORT NAME GETDODA

TYPE Resource function

DECLARATION

```
VRLEN GETDODA(datno,bufsiz,bufptr,mode)
COUNT datno;
LONG bufsiz;
VOID *bufptr;
COUNT mode;
```

DESCRIPTION **GetDODA** returns various forms of the records schema (if any) assigned to the data file as a resource into the buffer pointed to by *bufptr*. *bufsiz* specifies the length of the buffer. *mode* indicates the form of the return information. The value returned by **GetDODA** is the actual length of the information requested. If *bufsiz* is too small, **GetDODA** still returns the actual size, but no data is transferred into the buffer. Another call to **GetDODA** with a proper size buffer is required.

mode may take on one of the following values.

Symbolic Constant	Value	Interpretation
SCHEMA_MAP	1	return the record schema map.
SCHEMA_NAMES	2	return the record schema symbolic names.
SCHEMA_MAPandNAMES	3	return the map and symbolic names.
SCHEMA_DODA	4	return the info in the form of an array of DATOBJ's (a DODA).

If SCHEMA_MAP is requested, then a map as described on page PRG D-1 is returned in the buffer. If SCHEMA_NAMES is requested, then only the names in the form described on page PRG D-2 are returned in the buffer. If SCHEMA_MAPandNAMES is requested, the buffer contains the schema map followed immediately by the schema name. The *maplen* member of the schema map (see page PRG D-2) is the offset from the beginning of the buffer to the names.

If SCHEMA_DODA is requested, the the map and names (if any) are converted to the form of a DODA where the *fadr* member of the DATOBJ structure is filled in with the offset of the field from the beginning of the record.

RETURN In the event of an error, GETDODA returns a zero value.

Value	Symbolic Constant	Explanation
22	FNUM_ERR	*datno* out of range.
26	FACS_ERR	*datno* is not active.
116	IMOD_ERR	invalid *mode* value.
401	RNON_ERR	resources not enabled.
408	RNOT_ERR	no schema information exists.
438	DZRO_ERR	no schema information exists

SEE ALSO **PutDODA**

GetFileInfo

Retrieve information from a file header.

SHORT NAME GETFIL

TYPE Low Level function

DECLARATION

```
LONG GetFileInfo(filno,mode);
COUNT filno;
COUNT mode;
```

DESCRIPTION

A variety of information about a file is stored in the file header. However, in c-tree Plus the header is not directly accessible.

GetFileInfo can be used to access certain items of information from the header of file *filno*. Use *mode* to specify the type of information to be returned. The following values can be used for *mode*:

RECLEN	return data record length (valid only for data files).
KEYLEN	return key length (valid only for key files).
LOGSIZ	return the logical file size in bytes. The logical file size represents the number of bytes that are taken up by actual data records, including deleted records.
PHYSIZ	return the physical file size in bytes. The physical size is the actual size of the file, including deleted records and file extension space not yet used.
FILTYP	return 0 if the file is a Fixed Length Data file, 1 if it is an Index file, or 2 if it is a Variable Length Data file.
FILMOD	return the file mode value.
REVMAP	returns the data file number associated with the key *filno* (ISAM files only).
KEYTYP	returns the key type.
NODSIZ	returns the index node size.

RETURN The return value depends on the *mode* used in the function call. The value returned is listed above. A value of -1L means that an error has occured.

Value	Symbolic Constant	Explanation
22	FNUM_ERR	*filno* out of range.
26	FACS_ERR	*filno* is not active.
116	IMOD_ERR	invalid *mode* value.

EXAMPLE

```
COUNT datfil;

printf("\nThe record length is
%l",GetFileInfo(datfil,RECLEN));
```

SEE ALSO **GetSymbolicNames**

GetGTEKey

Search index file for entry equal to or greater than target.

SHORT NAME GTEKEY

TYPE Low level index file function

DECLARATION

```
LONG GetGTEKey(keyno,target,idxval)
COUNT keyno;
VOID *target,*idxval;
```

DESCRIPTION **GetGTEKey** searches index file *keyno* for the first entry which is equal to or greater than the key value pointed to by *target*. If such an entry exists, then it is copied into the area pointed to by *idxval*.

GetGTEKey is frequently used to start searching an index file which accepts **duplicate keys**. **GetKey** cannot be used since the index entries have suffixes appended. If index file *keyno* accepts duplicate entries, then you must be careful to pad the target key value with null bytes to ensure that you find the entry with the smallest suffix.

RETURN If **GetGTEKey** is successful, it returns the data record position associated with the entry found in the index. If an error occurs or no such entry exists (either the index is empty or the target key value exceeds all entries in the index), then **GetGTEKey** returns a zero. After a call to **GETGTEKey** which returns a zero, check the value of uerr_cod: if uerr_cod is non-zero, an error condition was detected; otherwise no key value satisfied the search.

EXAMPLE

```
COUNT          keyno;
TEXT           target[24],*idxval;
LONG          *suffix;

/* Key length = 23.
 * Assuming duplicates are allowed, the last four bytes
 * are for the suffix.
 */

scanf("%19s",target);
suffix = target + 20;/*pointer to the last four bytes*/
*suffix = 0L;          /* set suffix to zero. Assumes no
          *               alignment restrictions. See
          *               CTEXMG.C for a method which
          *               avoids alignment problems.
          */

idxval = "12345678901234567890123";
```

```
            /* reserve space for idxval*/
if (GetGTEKey(keyno,target,idxval))
   printf("\nthe key value >= %.19s is %.19s",
     target,idxval);
```

LIMITATIONS

No check is made to determine if *idxval* points to a region sufficiently large to accept a key value from the index file. If the area is too small, either code or data will be clobbered.

Do not use the same pointer value for the *target* and *idxval* parameters.

SEE ALSO

GetKey, **GetGTKey** and **GetLTKey**.

GetGTERecord

Read the record with key greater than or equal to target.

SHORT NAME GTEREC

TYPE ISAM function

DECLARATION

```
COUNT GetGTERecord(keyno,target,recptr)
COUNT keyno;
VOID *target,*recptr;
```

DESCRIPTION

GetGTERecord reads the first data record whose key value is equal to or greater than the key value pointed to by *target* into the buffer area pointed to by *recptr*. If successful, the record becomes the current ISAM record for the data file associated with *keyno*.

GetGTERecord is frequently used to start searching a file via an index file which accepts duplicate keys. **GetRecord** cannot be used since the index entries have suffixes appended. If index file *keyno* accepts duplicate entries, then you must be careful to pad the target key value with null bytes to ensure that you find the entry with the smallest suffix. See the example for **GetGTEKey**. An alternative to **GetGTERecord** is **FirstInSet** which reads the first data record whose key value matches the significant bytes in a target key; it is not necessary to pad the *target* when using **FirstInSet**. The main difference between **GetGTERecord** and **FirstInSet** is that **GetGTERecord** does not require any match with the target.

If the associated data file has variable length records, then only the minimum record length portion (defined by *datlen* in the original call to **CreateDataFile**) is actually read into the buffer pointed to by *recptr*. If you wish to read the entire variable length record into the same or a different buffer, issue a call to **ReReadVRecord** after the call to **GetGTERecord**. Note that **ReReadVRecord** requires you to specify the size of the buffer area so that it can check if sufficient space is available.

Notice that you do not directly identify the data file number involved. The ISAM parameter file described in Chapter PRG 5, contains the correspondence between the index file number (*keyno*) and the associated data file.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful update of current ISAM record.
33	DNUL_ERR	*recptr* is NULL. No data file read performed.

Value	Symbolic Constant	Explanation
42	DLOK_ERR	Could not get lock on data record. No data file read performed.
101	INOT_ERR	No entry found equal to or greater than target key value.

EXAMPLE

```
LONG          cust_number;
COUNT         keyno;
TEXT         *recptr;

scanf("%ld",cust_number);
if (GetGTERecord(keyno,&cust_number,recptr))
 printf("\nUnsuccessful search. Code = %d on file %d.",
 isam_err,isam_fil);
```

LIMITATIONS

No check is made to determine if *recptr* points to a region sufficiently large to accept a data record. If the area is too small, either code or data will be clobbered.

SEE ALSO

GetRecord, **FirstInSet** and **ReReadVRecord**.

GetGTKey

Search index file for entry greater than target.

SHORT NAME GTKEY

TYPE Low level index file function

DECLARATION

```
LONG GetGTKey(keyno,target,idxval)
COUNT keyno;
VOID *target,*idxval;
```

DESCRIPTION **GetGTKey** searches index file *keyno* for the first entry which is strictly greater than the key value pointed to by *target*. If such an entry exists, then it is copied into the area pointed to by *idxval*.

RETURN If **GetGTKey** is successful, it returns the data record position associated with the entry found in the index. If an error occurs or no such entry exists (either the index is empty or the target key value exceeds all entries in the index), then **GetGTKey** returns a zero. After a call to **GetGTKey** which returns a zero, check the value of uerr_cod: if uerr_cod is non-zero, an error condition was detected; otherwise no key value satisfied the search.

EXAMPLE

```
COUNT   keyno;
TEXT    target[24],idxval[23];
LONG    recbyt;

scanf("%23s",target);
recbyt = GetGTEKey(keyno,target,idxval);
while (recbyt) {
   printf("\nIndex entry = %23s   Record # = %ld",
     idxval,recbyt);
   cpybuf(target,idxval,23);
   recbyt = GetGTKey(keyno,target,idxval);
}
```

LIMITATIONS No check is made to determine if *idxval* points to a region sufficiently large to accept a key value from the index file. If the area is too small, either code or data will be clobbered.

Do not use the same pointer value for the target and *idxval* parameters.

GetIFile

Retrieve an IFIL structure from a data file.

SHORT NAME GETIFIL

TYPE ISAM/Resource

DECLARATION

```
VRLEN GETIFIL(datno,bufsiz,bufptr)
COUNT datno;
LONG bufsiz;
VOID *bufptr;
```

DESCRIPTION **GetIFile** returns a complete IFIL structure into the buffer pointed to by *bufptr*. *bufsiz* contains the size of the buffer. The value returned by **GetIFile** is the actual length of the IFIL structure (and it subsidiary IIDX and ISEG structures which are included in the buffer). If *bufsiz* is too small, then **GetIFile** still returns the actual size, but no IFIL information is returned. In this case, call **GetIFile** again with a sufficiently large buffer.

It is permissible to cast the buffer pointer to a pointer of type IFIL (IFIL *).

RETURN If there is an error, **GetIFile** returns a zero value. The error code can be found in *isam_err*.

Value	Symbolic Constant	Explanation
401	RNON_ERR	resources not enabled.
408	RNOT_ERR	no IFIL information assigned to the file.
438	DZRO_ERR	no IFIL information assigned to the file

SEE ALSO **OpenFileWithResource**

GetKey

Search index file for entry equal to target value.

SHORT NAME EQLKEY

TYPE Low level index file function

DECLARATION

```
LONG GetKey(keyno,target)
COUNT keyno;
VOID *target;
```

DESCRIPTION **GetKey** searches index file number *keyno* for an entry equal to the key value pointed to by *target.*

RETURN If **GetKey** finds an exact match for *target,* then it returns the data record position associated with the entry. If an exact match is not found, or if an error occurs, then **GetKey** returns a zero, and the global variable uerr_cod is set to the appropriate value.

EXAMPLE

```
LONG          recbyt;
COUNT         keyno;
TEXT          target[11];

scanf("%10s %d",target,&keyno);
if (!(recbyt = GetKey(keyno,target))) {
   if (uerr_cod)
     printf("\nError while searching.");
   else
     printf("\nKey value not found.");
} else
   printf("\nKey value is in record at position %ld.",
     recbyt);
```

LIMITATIONS Recall that c-tree Plus will not pad the target key value. Be sure that *target* points to a completely determined key value.

SEE ALSO **GetGTEKey**, **GetGTKey** and **GetLTKey**.

GetLTEKey

Search index file for entry equal to or less than target.

SHORT NAME LTEKEY

TYPE Low level index file function

DECLARATION

```
LONG GetLTEKey(keyno,target,idxval)
COUNT keyno;
VOID *target,*idxval;
```

DESCRIPTION **GetLTEKey** searches index file *keyno* for the last entry which is equal to or less than the key value pointed to by *target*. If such an entry exists, then it is copied into the area pointed to by *idxval*.

RETURN If **GetLTEKey** is successful, it returns the data record position associated with the entry found in the index. If an error occurs or no such entry exists (either the index is empty or the target key value is less than all entries in the index), then **GetLTEKey** returns a zero. After a call to **GetLTEKey** which returns a zero, check the value of uerr_cod: if uerr_cod is non-zero, an error condition was detected; otherwise no key value satisfied the search.

EXAMPLE

```
COUNT          i,keyno;
TEXT           target[24],idxval[24];
TEXT          *suffix;

/* Key length = 23.
 * Assuming duplicates are allowed, the last four bytes
 * are for the suffix.
 */

scanf("%19s",target);
suffix = target+20;/* pointer to the last four bytes*/
for (i = 0;i++ < 4;) /*set suffix to highest possible*/
   *suffix++ = '\377';  /* value                      */
if (GetLTEKey(keyno,target,idxval))
   printf("\nthe key value <= %.19s is %.19s",
     target,idxval);
```

LIMITATIONS No check is made to determine if *idxval* points to a region sufficiently large to accept a key value from the index file. If the area is too small, either code or data will be clobbered.

Do not use the same pointer value for the *target* and *idxval* parameters.

SEE ALSO **GetKey**, **GetGTEKey**, **GetGTKey** and **GetLTKey**.

GetLTERecord

Read the record with key less than or equal to target.

SHORT NAME LTEREC

TYPE ISAM function

DECLARATION

```
COUNT GetLTERecord(keyno,target,recptr)
COUNT keyno;
VOID *target,*recptr;
```

DESCRIPTION

GetLTERecord reads the last data record whose key value is equal to or less than the key value pointed to by *target* into the buffer area pointed to by *recptr*. If successful, the record becomes the current ISAM record for the data file associated with *keyno*.

GetLTERecord may be used to start searching a file in reverse order. Also see **LastInSet** for this purpose.

If the associated data file has variable length records, then only the minimum record length portion (defined by *datlen* in the original call to **CreateDataFile**) is actually read into the buffer pointed to by *recptr*. If you wish to read the entire variable length record into the same or a different buffer, issue a call to **ReReadVRecord** after the call to **GetLTERecord**. Note that **ReReadVRecord** requires you to specify the size of the buffer area so that it can check if sufficient space is available.

Notice that you do not directly identify the data file number involved. The ISAM parameter file described in Chapter PRG 5, contains the correspondence between the index file number (*keyno*) and the associated data file.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful update of current ISAM record.
33	DNUL_ERR	*recptr* is NULL. No data file read performed.
42	DLOK_ERR	Could not get lock on data record. No data file read performed.
101	INOT_ERR	No entry found equal to or less than target key value.

EXAMPLE

```
LONG          cust_number;
COUNT         keyno;
TEXT         *recptr;

scanf("%ld",cust_number);
if (GetLTERecord(keyno,&cust_number,recptr))
 printf("\nUnsuccessful search. Code = %d on file %d.",
     isam_err,isam_fil);
```

LIMITATIONS

No check is made to determine if *recptr* points to a region sufficiently large to accept a data record. If the area is too small, either code or data will be clobbered.

SEE ALSO

GetRecord, **LastInSet** and **ReReadVRecord**.

GetLTKey

Search index file for entry less than target.

SHORT NAME LTKEY

TYPE Low level index file function

DECLARATION

```
LONG GetLTKey(keyno,target,idxval)
COUNT keyno;
VOID *target,*idxval;
```

DESCRIPTION **GetLTKey** searches index file *keyno* for the last entry which is strictly less than the key value pointed to by *target*. If such an entry exists, then it is copied into the area pointed to by *idxval*.

RETURN If **GetLTKey** is successful it returns the data record position associated with the entry found in the index. If an error occurs or no such entry exists (either the index is empty or the target key value is smaller than all entries in the index), then **GetLTKey** returns a zero. After a call to **GetLTKey** which returns a zero, check the value of uerr_cod: if uerr_cod is non-zero, an error condition was detected; otherwise no key value satisfied the search.

EXAMPLE

```
COUNT   keyno;
TEXT    target[24],idxval[23];

scanf("%23s",target);
if (GetLTKey(keyno,target,idxval))
   printf("\nIndex entry %.23s < %s",idxval,target);
```

LIMITATIONS No check is made to determine if *idxval* points to a region sufficiently large to accept a key value from the index file. If the area is too small, either code or data will be clobbered.

Do not use the same pointer value for the *target* and *idxval* parameters.

SEE ALSO **GetKey**, **GetGTKey** and **GetGTEKey**.

GetRecord

Get data record with key equal to target value.

SHORT NAME

EQLREC

TYPE

ISAM function

DECLARATION

```
COUNT GetRecord(keyno,target,recptr)
COUNT keyno;
VOID *target,*recptr;
```

DESCRIPTION

GetRecord searches index file number *keyno* for an entry which equals the key value pointed to by *target.* If an exact match is found, **GetRecord** reads the associated data record into the buffer area pointed to by *recptr,* and this record becomes the current ISAM record. If no match is found or an error occurs, the current ISAM record remains unchanged.

If the associated data file has variable length records, then only the minimum record length portion (defined by *datlen* in the original call to **CreateDataFile**) is actually read into the buffer pointed to by *recptr.* If you wish to read the entire variable length record into the same or a different buffer, issue a call to **ReReadVRecord** after the call to **GetRecord**. Note that **ReReadVRecord** requires you to specify the size of the buffer area so that it can check if sufficient space is available.

Notice that you do not directly identify the data file number involved. The ISAM parameter file described in Chapter PRG 5 contains the correspondence between the index file number *(keyno)* and the associated data file.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful update of current ISAM record.
33	DNUL_ERR	*recptr* is NULL. No data file read performed.
42	DLOK_ERR	Could not get lock on data record. No data file read performed.
101	INOT_ERR	No match for target key value found.

EXAMPLE

```
COUNT          part_idx;
TEXT           part_key[25];
struct invd    {
   TEXT        delflg;
   LONG        part_no;
   TEXT        part_name[60];
   LONG        on_hand;
   LONG        on_order;
} recbuf;

printf("\nEnter part name: ");
scanf("%24s",part_key);
switch (GetRecord(part_idx,part_key,&recbuf)) {

case NO_ERROR:
   printf("\nParts on order for %s = %ld",
           recbuf.part_name,recbuf.on_order);
   break;

case DLOK_ERR:
   printf("\nCould not get record lock.");
   break;

case INOT_ERR:
   printf("\nPart name not on file.");
   break;

default:
   printf("\nProgramming Error (code = %d file = %d.",
     isam_err,isam_fil);
}
```

LIMITATIONS

No check is made to determine if *recptr* points to a region sufficiently large to accept a data record. If the area is too small, either code or data will be clobbered.

SEE ALSO

GetGTERecord and **ReReadVRecord**.

GetResource

Get Resource record from data file.

SHORT NAME GETRES

TYPE Resource function.

DECLARATION

```
LONG GetResource(datno, resptr, bufptr, bufsiz,
                 resmode);
COUNT datno;
VOID  *resptr;
VOID  *bufptr;
VRLEN bufsiz;
COUNT resmode;
```

DESCRIPTION

GetResource is used to retrieve a Resource from data file *datno*. A long integer corresponding to the file address (byte offset) of the Resource is returned by the function, and the value of the Resource is placed in the buffer pointed to by *bufptr*. *bufsiz* is the size of the buffer area. Remember that the buffer will contain the entire Resource Data Block (see **AddResource** for information).

resmode is used to specify the method of searching for the Resource. *resptr* points to the identifying information to be used in the search. The following *resmode* choices are available:

RES_TYPNUM	find the Resource with the matching Type and Number. *resptr* points to a resource Type followed by a resource Number.
RES_TYPE	find the Resource with the matching Type and lowest available Number. *resptr* points to the resource Type.
RES_FIRST	find the Resource with the lowest available Type and Number. *resptr* is not used.
RES_NAME	find the first Resource with a matching Name (remember that c-tree cannot guarantee unique Resource Names). *resptr* points to a buffer that contains the Name to search for.
RES_POS	read the Resource at a given record address (which must must have been obtained in a prior call to **UpdateResource**). *resptr* points to a long integer with the record address to be read.

In addition you can OR the *resmode* with the value RES_LOCK. This will make **GetResource** ask for a low level exclusive lock on the matching Resource record.

This lock is released if you make a call to **UpdateResource** or **DeleteResource** for the same Resource, or with a call to **LockData**. It is important to note that Resource locks are **NOT** released by explicit or implicit calls to **LockISAM**.

RETURN

If no error occurs, **GetResource** returns the byte offset for the Resource record. Otherwise, **GetResource** returns a zero and uerr_cod is set as follows:

Value	Symbolic Constant	Explanation
0	NO_ERROR	No error.
42	DLOK_ERR	a lock cannot be placed on this record (only if RES_LOCK enabled).
48	FMOD_ERR	*datno* does not have Resources enabled.
401	RNON_ERR	*datno* does not have Resources enabled.
403	RZRO_ERR	empty resource ID
404	RBUF_ERR	*bufsiz* is smaller than the size of the information to be returned by **GetResource**.
406	RCSE_ERR	Invalid *resmode* value.
407	RRED_ERR	Attempt to get non-resource info. The byte offset provided with RES_POS mode does not point to a valid resource record.
408	RNOT_ERR	Resource not found. No resource is found that matches the values specified by *resmode*.

EXAMPLE

```
#include "ctstdr.h"
#include "ctdecl.h"

structure {                          /* Resource Data Block structure */
   ULONG  resource_type;
   ULONG  resource_number;
   TEXT   variable_info[1016];    /* This holds the Resource Name
                                     and Data*/
} my_resource;

my_resource.resource_type    = 0x10001L;     /* 65537 */
my_resource.resource_number  = 0xffffffffL;  /* c-tree to assign
                                                number*/
strcpy(my_resource.variable_info,"M&!resource");
strcat(my_resource.variable_info,"Actual Resource Data");

if (AddResource(datno,&my_resource,
```

```
        (VRLEN)(8 + strlen(my_resource.variable_info))) == 0 )
    {
            printf("\nThe resource has been assigned number %ld",
                my_resource.resource_number);
            printf("\nThe resource is located at byte offset %ld.",
                GetResource(datno,"MY!resource",&my_resource,1024L);
    } else
            printf("\nCould not add resource. Error #%d",uerr_cod);
```

SEE ALSO **UpdateResource**, **AddResource**, **DeleteResource**.

GetSerialNbr

Get current data file sequence number.

SHORT NAME SERIALNUM

TYPE Low level data file function

DECLARATION

```
LONG GetSerialNbr(datno)
COUNT datno;
```

DESCRIPTION **GetSerialNbr** returns the current sequence number for data file *datno*. Each time **NewData** or **NewVData** are called, the sequence number is incremented. (Calls to **AddRecord** and **AddVRecord** automatically invoke calls to **NewData** and **NewVData**, respectively.) This sequence number is used in ISAM applications when a key segment mode of 3 is used (see Chapter PRG 5).

RETURN **GetSerialNbr** returns a long integer containing the current sequence number. If an error occurs (such as data file not in use), the **GetSerialNbr** returns a zero value.

EXAMPLE

```
COUNT datno;
if (OpenFile(datno,"MYDATA.DAT",(VIRTUAL | EXCLUSIVE)){
   printf("\nCould not open data file with error %d",
            uerr_cod);
   exit(2);
}
printf("\n MYDATA has had NewData called %ld times.",
   GetSerialNbr(datno));
exit(0);
```

GetSymbolicNames

Retrieve symbolic information about file.

SHORT NAME GETNAM

TYPE Low Level function

DECLARATION

```
LONG GetSymbolicNames(filno,nambuf,buflen,mode);
COUNT filno;
TEXT *nambuf;
VRLEN buflen;
COUNT mode;
```

DESCRIPTION **GetSymbolicNames** can be used to access certain items of information that relates to the file *filno*. Use *mode* to specify the type of information to be returned. The information is returned at *nambuf*. The following values can be used for *mode*:

- FILNAM - return the name of the file.
- FRSFLD - return the first **r-tree** data field symbolic name.
- LSTFLD - return the last **r-tree** data field symbolic name.
- IDXNAM - return the **r-tree** index symbolic name.

RETURN The return value will be NO_ERROR (0) if no error has occured. The information will be returned at *nambuf*. Otherwise, an error code will be returned.

Value	Symbolic Constant	Explanation
0	NO_ERROR	no error has occured.
22	FNUM_ERR	*keyno* out of range.
26	FACS_ERR	*keyno* is not active.
116	IMOD_ERR	invalid *mode* value.

EXAMPLE

```
COUNT filenum;
TEXT  buffer[128];

if (GetSymbolicNames(filnum,buffer,128L,FILNAM) ==
        NO_ERROR)
   printf("\nFile Name is %s",buffer);
```

SEE ALSO **GetFileInfo**.

GetTempFileName

Create a file name that is guaranteed to be unique.

SHORT NAME TMPNAME

TYPE Low level function

DECLARATION

```
COUNT GetTempFileName(bufptr,bufsiz);
VOID *bufptr;
VRLEN bufsiz;
```

DESCRIPTION **GetTempFileName** will create a file name that is guaranteed to be unique for you to use as a temporary file. The name will be placed in the buffer pointed to by *bufptr*. The size of the buffer is sent in *bufsiz*.

RETURN

Value	Symbolic Constant	Explanation
153	VBSZ_ERR	buffer size is too small for name.
155	SYST_ERR	system fails to make a name.

EXAMPLE

```
COUNT datafil;
TEXT  file_name[20];

if (GetTempFileName(file_name,20L)) return(uerr_cod);
CreateDataFile(datafil,file_name,128,4096,
       PERMANENT | EXCLUSIVE | FIXED);
```

InitCTree

Initialize file structures and index buffer area.

SHORT NAME INTREE

TYPE Low level file function

DECLARATION

```
COUNT InitCTree(bufs,fils,sect)
COUNT bufs,fils,sect;
```

DESCRIPTION **InitCTree** acquires the memory space necessary for the data and index file structures as well as the space for the index file buffers. Finally, it initializes this memory area.

The **InitCTree** parameters are defined as follows:

Parameter	Interpretation
bufs	The number of index file buffers. Minimum of three required.
fils	The maximum number of data files and indices to be opened simultaneously. Each index, whether it is a member of an index file or in a file by itself counts towards this value.
sect	The number of node sectors. Minimum of one required. *sect* multiplied by 128 equals the index node size.

In a single user system, increasing *bufs* reduces the number of disk accesses necessary to retrieve a key value. Beyond about six buffers per active index, the marginal return from additional buffers is very small. Note that the buffers are shared across all the index files via a "least recently used" scheme.

In non-server multi-user systems (FPUTFGET disk I/O protocol), the need to force updates back to the disk implies that local buffering is of little value. Rather, the shared hard disk should be cached in order to achieve global buffering of the index file. In a non-server multi-user system, *bufs* should be set around six, regardless of the number of index files in use.

sect multiplied by 128 determines how many bytes are available in each B- Tree node. When an index file is created, the node size is set by this product. When an index file is opened, a check is made to be sure that the current product (*sect* * 128) is at least as large as when the index was created. That is, you can open an index file if *sect* has been increased since the time of creation, but not if it has been reduced.

NOTE: The most common value used for *sect* is four which yields a node size of 512 bytes. The next most common value for *sect* is eight.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful call to **InitCTree**.
10	SPAC_ERR	Too much space required. Reduce *bufs*.
11	SPRM_ERR	Either *bufs* < 3, *sect* < 1, or *fils* < 0
125	IINT_ERR	**InitCTree** already called.

EXAMPLE

```
if (InitCTree(6,7,4))
   printf("\nCould not initialize c-tree.");
else {
   printf("\nc-tree initialized:\n\t6 buffers, ");
   printf("7 files and 512 byte nodes.");
}
```

SEE ALSO **CreateDataFile**, **CreateIndexFile**, **CreateIndexMember** **InitCTreeXtd** and **InitISAM**.

InitCTreeXTD

Initialize file structures and index buffer area (extended version).

SHORT NAME INTREEX

TYPE Extended low level file function

DECLARATION

```
COUNT
InitCTreeXtd(bufs,fils,sect,dbufs,userprof,userid,
                          userword,servname)
COUNT bufs,fils,sect,dbufs;
TEXT *userprof;
TEXT *userid;
TEXT *userword;
TEXT *servname;
```

InitCTreeXtd is a variation of **InitCTree** that permits the use of the FairCom Server's security system, as well as setting the user profile mask. Please read the description of **InitCTree** before proceeding with this description, as we will only describe the extended features here.

bufs and *dbufs* specify the number of buffers for index and data file I/O respectively. Generally you will want to set *bufs* and *dbufs* to the same value. If you have a program that will be primarily using index file manipulations, with very little work with data files, you can make *dbufs* smaller to save buffer space. In a Server environment *bufs*, *dbufs*, and *sect* are ignored. They are set when the Server is configured.

userprof is the **user profile mask**. If you set this to the value USERPRF_NTKEY you will **disable** the **automatic TransformKey** feature. See the function description for **TransformKey** for more information. If you do not wish to disable this feature, set *userprof* to zero.

userid is a pointer to a buffer containing your user ID. See page PRG 12-2 for more information on user ID's.

userword is a pointer to a buffer containing your user password. See page PRG 12-2 for more information on user passwords.

servname is a pointer to a server name if you are going to use a server other than the default server, in a multi-server environment. See the Server documentation for more information.

RETURN The following error codes may be seen in addition to those shown for **InitCTree**.

Value	Symbolic Constant	Explanation
450	LUID_ERR	Invalid user ID.
451	LPWD_ERR	Invalid user password.
453	NSRV_ERR	Invalid server name.

InitISAM

Initialize file structures and index buffer area.

SHORT NAME INTISAM

TYPE ISAM level function

DECLARATION

```
COUNT InitISAM(bufs,fils,sect)
COUNT bufs,fils,sect;
```

DESCRIPTION **InitISAM** acquires the memory space necessary for the data and index file structures as well as the space for the index file buffers. Finally, it initializes this memory area, and prepares for incremental ISAM creates and opens. **InitISAM** should only be used when no ISAM parameter is to be active.

Before any other ISAM functions can be used, one of **InitISAM**, **OpenISAM** or **CreateISAM** must be called. Further, these are mutually exclusive; only one of these routines can be called at a time. Once one of these has been called, you must call **CloseISAM** before invoking one of these three functions again.

The **InitISAM** parameters are defined as follows:

Parameter	Interpretation
bufs	The number of index file buffers. Minimum of three required.
fils	The maximum number of data files and indices to be opened simultaneously. Each index, whether it is a member of an index file or in a file by itself counts toward the *fils* parameter.
sect	The number of node sectors. Minimum of one required. *sect* multiplied by 128 equals the index node size.

In a single user system, increasing *bufs* reduces the number of disk accesses necessary to retrieve a key value. Beyond about six buffers per active index file, the marginal return from additional buffers is very small. Note that the buffers are shared across all the index files via a "least recently used" scheme.

In non-server multi-user systems, the need to force updates back to the disk implies that local buffering is of little value. Rather, the shared hard disk should be cached in order to achieve global buffering of the index file. In a non-server multi-user system, *bufs* should be set around six, regardless of the number of index files in use.

sect * 128 determines how many bytes are available in each B- Tree node. When an index file is created, the node size is set by this product. When an index file is opened, a check is made to be sure that the current product (*sect* * 128) is at least as large as when the index was created. That is, you can open an index file if *sect* has been increased since the time of creation, but not if it has been reduced.

NOTE: The most common value used for *sect* is four which yields a node size of 512 bytes. The next most common value for *sect* is eight.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful call to **InitISAM**.
10	SPAC_ERR	Too much space required. Reduce *bufs*.
11	SPRM_ERR	Either *bufs* < 3
104	IFIL_ERR	Too many ISAM data files and indices. Change MAX-FIL in *ctoptn.h*.
125	IINT_ERR	c-tree Plus already initialized. Call **CloseISAM** before recalling **InitISAM**.
182	IKSR_ERR	*fils* parameter does not accommodate three temporary r-tree files. Increase MAXFIL.

EXAMPLE

```
if (InitISAM(6,7,4)) {
   printf("\nCould not initialize c-tree. Error #%d\n",
     isam_err);
   exit(2);
}
```

SEE ALSO **OpenIFile, CreateIFile, OpenISAM, CreateISAM InitISAMXtd** and **CloseISAM.**

InitISAMXtd

Initialize file structures and index buffer area (extended version).

SHORT NAME INTISAMX

TYPE Extended ISAM level function

DECLARATION

```
COUNT InitISAMXtd(bufs,fils,sect,dbufs,userprof,userid,
                  userword,servname)
COUNT bufs,fils,sect,dbufs;
TEXT *userprof;
TEXT *userid;
TEXT *userword;
TEXT *servname;
```

InitISAMXtd is a variation of **InitISAM** that permits the use of the FairCom Server's security system, as well as setting the user profile mask. Please read the description of **InitISAM** before proceeding with this description, as we will only describe the extended features here.

bufs and *dbufs* specify the number of buffers for index and data file I/O respectively. Generally you will want to set *bufs* and *dbufs* to the same value. If you have a program that will be primarily using index file manipulations, with very little work with data files, you can make *dbufs* smaller to save buffer space. In a Server environment *bufs*, *dbufs*, and *sect* are ignored. They are set when the Server is configured.

userprof is the **user profile mask**. If you set this to the value USERPRF_NTKEY you will **disable** the **automatic TransformKey** feature. See the function description for **TransformKey** for more information. If you do not wish to disable this feature, set *userprof* to zero.

userid is a pointer to a buffer containing your user ID. See page PRG 12-2 for more information on user ID's.

userword is a pointer to a buffer containing your user password. See page PRG 12-2 for more information on user passwords.

servname is a pointer to a server name if you are going to use a server other than the default server, in a multi-server environment. See the Server documentation for more information.

RETURN The following error codes may be seen in addition to those shown for **InitISAM**.

Value	Symbolic Constant	Explanation
450	LUID_ERR	Invalid user ID.
451	LPWD_ERR	Invalid user password.
453	NSRV_ERR	Invalid server name.

KeyAtPercentile

Find key value located at about the given percentile.

SHORT NAME FRCKEY

TYPE Low level index file function

DECLARATION

```
LONG KeyAtPercentile(keyno,idxval,percnt)
COUNT keyno;
VOID *idxval;
COUNT percnt;
```

DESCRIPTION

KeyAtPercentile finds the key value located, approximately, at the percentile specified by *percnt*. The key value is copied into the area pointed to by *idxval*. percnt should be in the range from zero (0) to one-hundred (100).

KeyAtPercentile is very efficient since it does not traverse all of the key values in order to determine the key value located at the specified percentile. However, **KeyAtPercentile** is only approximate because it assumes that the key values are uniformly distributed among all of the b-tree leaf nodes.

KeyAtPercentile can be used to support scroll bar positioning (found in many windowing environments) in those cases when the position must be maintained in key sequential order.

RETURN

If **KeyAtPercentile** is successful, it returns the data record position associated with the key value in the index which is located, approximately, at the specified percentile value. If an error occurs or the index is empty, then **KeyAtPercentile** returns a zero. After a call to **KeyAtPercentile** which returns a zero, check the value of uerr_cod: if uerr_cod is non-zero, an error condition was detected; otherwise, the index is empty.

EXAMPLE

```
COUNT       keyno;
TEXT        idxval[24];


if (KeyAtPercentile(keyno,idxval,75))
   printf("\n value at the 75th percentile is %.23s",
     idxval);
else if (uerr_cod)
   printf("\n error %d occurred",uerr_cod);
else
   printf("\n the index is empty");
```

LIMITATIONS

Since **KeyAtPercentile** is only approximate, it may be necessary in scroll bar applications to ensure that the result of **KeyAtPercentile** is consistent with the

current location. For example, you may want to detect the direction of the scroll bar motion and ensure that the key value returned is consistent.

While **KeyAtPercentile** is approximate, it is self-consistent. As you increase (decrease) percnt, the associated key values will increase (decrease) in value.

SEE ALSO **EstimateKeySpan**.

LastInSet

Read the last data record in the set defined by target.

SHORT NAME LSTSET

TYPE ISAM function

DECLARATION

```
COUNT LastInSet(keyno,target,recptr,siglen)
COUNT keyno;
VOID *target,*recptr;
COUNT siglen;
```

DESCRIPTION

LastInSet reads the last data record in the set of data records whose keys match the target key in the first *siglen* bytes. If successful, this record becomes the current ISAM record for the associated data file. A successful **LastInSet** defines a current key value set, and subsequent calls to **PreviousInSet** and **NextInSet** will read the other records in the set (i.e., those records whose keys also match the first *siglen* bytes of *target*). If an error occurs or no key value matches the target, the current ISAM record is not updated, and no key value set is defined.

If the associated data file has variable length records, then only the minimum record length portion (defined by *datlen* in the original call to **CreateDataFile**) is actually read into the buffer pointed to by *recptr*. If you wish to read the entire variable length record into the same or a different buffer, issue a call to **ReReadVRecord** after the call to **LastInSet**. Note that **ReReadVRecord** requires you to specify the size of the buffer area so that it can check if sufficient space is available.

Notice that you do not directly identify the data file number involved. The ISAM parameter file described in Chapter PRG 5 contains the correspondence between the index file number (*keyno*) and the associated data file.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful update of current ISAM record and definition of key value set.
33	DNUL_ERR	*recptr* is NULL. No data file read performed.
42	DLOK_ERR	Could not get lock on data record. No data file read performed.
101	INOT_ERR	No matching key value found.
119	SKTY_ERR	*keyno* must reference a non-numeric key type.

EXAMPLE

```
COUNT          keyfil;
TEXT           target[24],recbuf[320];

printf("\nEnter set value:");
scanf("%23s",target);
LastInSet(keyfil,target,recbuf,strlen(target));
while (isam_err == NO_ERROR) {
   process_data();
   PreviousInSet(keyfil,recbuf);
}
```

LIMITATIONS

LastInSet can only be called for keys which are scanned from left to right and are non-numeric. Therefore, key types 1, 2 and 3 are not permitted.

No check is made to determine if *recptr* points to a region sufficiently large to accept a data record. If the area is too small, either code or data will be clobbered.

SEE ALSO

FirstInSet, **NextInSet**, **PreviousInSet**, **PositionSet** and **ChangeSet**.

LastKey

Find the last entry in an index file.

SHORT NAME LSTKEY

TYPE Low level index file function

DECLARATION

```
LONG LastKey(keyno,idxval);
COUNT keyno;
VOID *idxval;
```

DESCRIPTION **LastKey** searches index file *keyno* for the last entry in the index. If successful, it copies the last index entry into the space pointed to by *idxval*.

RETURN **LastKey** returns the data record position associated with the last entry in the index file. If the index is empty or an error is detected, then **LastKey** returns a zero. After a call to **LastKey** which returns a zero, check the value of uerr_cod: if uerr_cod is nonzero, an error condition was detected; otherwise the index is empty.

EXAMPLE

```
TEXT idxval[10];
COUNT  keyno;

if (LastKey(keyno,idxval))
   printf("\nLast index entry = %.10s");
else {
   printf("\nEither index is empty or error occurred.");
   if (uerr_cod)
     printf("\Error = %d",uerr_cod);
   else
     printf("\Index is empty.);
}
```

LIMITATIONS No check is made to determine if *idxval* points to a region sufficiently large to accept a key value from the index file. If the area is too small, either code or data will be clobbered.

SEE ALSO **FirstKey**.

LastRecord

Read the last data record.

SHORT NAME LSTREC

TYPE ISAM function

DECLARATION

```
COUNT LastRecord(filno,recptr)
COUNT filno;
VOID *recptr;
```

DESCRIPTION

If **LastRecord** is called with a *filno* corresponding to an index, then **LastRecord** reads the last data record, based on key sequential order for index number *filno*. If it is called with a *filno* corresponding to a fixed length data file, then **LastRecord** reads the last record in physical sequential order, automatically skipping over deleted records. If successful, the last record becomes the current ISAM record for the associated data file. If an error occurs, the current ISAM record remains the same.

NOTE: *filno* cannot be a data file number if any of the following are true:

- the data file is set up for variable length records;
- the data file is a member of a Superfile;
- the data file has Resources enabled (the default state).

In any of the above cases the error FMOD_ERR will be returned.

If *filno* is a key number whose associated data file has variable length records, then only the minimum record length portion (defined by *datlen* in the original call to **CreateDataFile**) is actually read into the buffer pointed to by *recptr*. If you wish to read the entire variable length record into the same or a different buffer, issue a call to **ReReadVRecord** after the call to **LastRecord**. Note that **ReReadVRecord** requires you to specify the size of the buffer area so that it can check if sufficient space is available.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful update of current ISAM record.
33	DNUL_ERR	*recptr* is NULL. No data file read performed.
42	DLOK_ERR	Could not get lock on data record. No data file read performed.

Value	Symbolic Constant	Explanation
48	FMOD_ERR	**LastRecord** called with a data file number that is: a variable length data file; a member of a superfile; has Resources enabled.
101	INOT_ERR	Empty file

EXAMPLE

```
COUNT  keyfil;
TEXT recbuf[320];

   if (!LastRecord(keyfil,recbuf))
     printf("\nSuccessful LastRecord.");
```

LIMITATIONS

No check is made to determine if *recptr* points to a region sufficiently large to accept a data record. If the area is too small, either code or data will be clobbered.

SEE ALSO

FirstRecord and **ReReadVRecord**.

LoadKey

Add key value to index file in presorted order.

SHORT NAME LOADKEY

TYPE Low level index file function

DECLARATION

```
COUNT LoadKey(keyno,target,recbyt,typadd)
COUNT   keyno;
VOID   *target;
LONG    recbyt;
COUNT   typadd;
```

DESCRIPTION

LoadKey inserts the key value pointed to by *target* and the associated data record position *recbyt* into index file number *keyno*. Repeated calls to **LoadKey** provide an exceptionally fast method to add key values which have already been sorted. The sorted keys must be added to an empty index or all of the sorted keys must be greater than the existing entries.

Calls to **LoadKey** should not be intermixed for different indices.

typadd signals the beginning, middle and end of the calls to **LoadKey** as shown below:

typadd Value	Symbolic Constant	Sequence
0	FRSADD	The first call to **LoadKey** for a given index must have *typadd* set to zero.
1	NXTADD	Subsequent calls to **LoadKey** must have *typadd* set to one.
2	BLDADD	After all the key values have been added, one additional call is made to **LoadKey** with *typadd* set to two. This last call causes **LoadKey** to complete the b-tree structure for the index, and should be made with a NULL pointer for *target*.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful key value insertion.

Value	Symbolic Constant	Explanation
48	FMOD_ERR	**LoadKey** called for a data file instead of index file.
58	KLOD_ERR	**LoadKey** called for index which has not been initialized by a call with *typadd* = FRSADD, or an attempt to mix calls for different indices.
59	KLOR_ERR	key value is out of order and has not been added to index: *target* is less than previous entry or not greater than existing key values. Be careful with duplicate keys, they must be presented in record position order when the keys are identical. It is permissible to continue calling **LoadKey** after a KLOR_ERR.

EXAMPLE

```
TEXT          target[11];
POINTER       recbyt,trials;
COUNT         name_key,mode;
FILE      *fp;
trials = 0L;
mode   = FRSADD;
while (fscanf(fp,"%s %ld",target,&recbyt) == 2) {
   if (LoadKey(name_key,target,recbyt,mode))
     break;
   if (trials++)
     mode = NXTADD;
}

if (uerr_cod || LoadKey(name_key,NULL,0L,BLDADD))
   printf("\nerror during LoadKey (%d) on trial #%ld",
     uerr_cod,trials);
```

LIMITATIONS

LoadKey does not pad key values. Therefore, if you pass a key value which is not completely specified to its full length, **LoadKey** will effectively pad the value with whatever "garbage" follows the key value in memory.

Key values added to an index which supports duplicate keys (keydup = 1 in **CreateIndexFile** or **CreateIndexMember**) have their last four bytes overwritten by the associated data record position. In this way, identical key values become distinguishable.

SEE ALSO

AddKey.

LockData

Lock / unlock data records.

SHORT NAME LOKREC

TYPE Low level data file function

DECLARATION

```
COUNT LockData(datno,lokmod,recbyt)
COUNT datno,lokmod;
LONG recbyt;
```

DESCRIPTION

LockData provides low level lock and unlock control over fixed and variable length data records. **LockData** can also be used with ISAM files in order to relax the constraints of the "two-phase" protocol (see Chapter PRG 6). Since c-tree Plus is based on write locks, a locked record can be read by other processes as long as they do not attempt their own write lock on the record.

lokmod specifies whether to attempt a lock or unlock operation. A zero value for *lokmod* (FREE) indicates an unlock operation. A value of 4 (READREC) for *lokmod* indicates a read lock operation. Read locks may be acquired by any number of users on the same data record at the same time. A read lock ensures that no other user can acquire a write lock on the record. A value of 2 (ENABLE) for *lokmod* indicates a write lock operation. Only one process may obtain a write lock on a record. A value of 3 (ENABLE_BLK) is the same as ENABLE, but a process unable to get a lock sleeps until the lock is available, instead of returning DLOK_ERR. However, if a dead lock is detected, DEAD_ERR is returned.

NOTE: many systems do not support read locks. If read locks are not supported, c-tree Plus returns a no error condition (i.e., the read lock is not attempted, but no error is reported). The FairCom Server *DOES* support read locks and write locks.

recbyt specifies the beginning byte offset for the record lock/unlock operation. c-tree Plus automatically provides the data file record length for the number of bytes to be locked. In the case of a variable length file, c-tree Plus only applies the lock to the minimum, fixed-length portion of the record. (In any event, c-tree Plus always asks for at least a one-byte lock.)

In single-user applications (i.e., the disk I/O protocol is not set to FPUTFGET), **LockData** always returns successfully.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful lock/unlock operation.
41	UDLK_ERR	Could not unlock data record. Either data record was not originally locked; or, if a dummy lock file is required to implement write-locks, then be sure that dummy lock file has a file mode of three.
42	DLOK_ERR	Record is already locked. (In some operating systems such as MS-DOS, you cannot attempt to relock a record you have already locked.)
48	FMOD_ERR	**LockData** called for an index file.
86	DEAD_ERR	Waiting for a write lock would cause a dead lock.

EXAMPLE

```
LONG pntr;
COUNT datno;
TEXT recbuf[147];
if ((pntr = NewData(datno)) == 0L ||
   fill_buffer(recbuf) != NO_ERROR ||
   WriteData(datno,pntr,recbuf) ||
   LockData(datno,FREE,pntr))
     printf("\nCould not write new record (%d)",uerr_cod);
/* recall that NewData always acquires a lock on
    a new record */
```

SEE ALSO

LockISAM, **NewData** and **NewVData**.

LockISAM

Enable, free or suspend data record locks.

SHORT NAME LKISAM

TYPE ISAM function

DECLARATION

```
COUNT LockISAM(lokmod)
COUNT lokmod;
```

DESCRIPTION **LockISAM** manages the data record locks during multiple-user ISAM file handling. The LockISAM options, defined in *ctport.h*, are:

lokmod Value	Symbolic Constant	Explanation
0	FREE	Free all data record locks held by this program and cancel the ENABLE mode.
1	RESET	Equivalent to first calling **LockISAM** in the FREE mode followed by a call in the ENABLE mode.
2	ENABLE	Once **LockISAM** is called with *lokmod* set to ENABLE, all subsequent data record reads attempt to get a write lock before proceeding; write locks prevent more than one process from updating the record at the same time since only one process may acquire a write lock on a record. Any number of processes may still read the record.
3	ENABLE_BLK	Same as ENABLE, but a process unable to get a write lock is "put to sleep" until the lock is available.
4	READREC	Once **LockISAM** is called with *lokmod* set to READREC, all subsequent data record reads attempt to get a read lock before proceeding; read locks prevent another process from acquiring a write lock on the record. Any number of processes may read locks on the same record.
5	SUSPEND	Temporarily suspend the attempt to get a lock on each ISAM record read.
6	RESTORE	Restore the ENABLE mode after a previous call to suspend data record locks.
7	RESTRED	Restore the READREC mode after a previous call to suspend data record locks.
8	RESTORE_BLK	Restore the ENABLE_BLK mode after a previous call to suspend data record locks.

To ensure that consistent updates are made in multiple-user systems, each program should ENABLE data record locks prior to updating data, and hold these locks (which might apply to several different data files) until the operations associated with those data records are completed. At that time, a call to **LockISAM** with the FREE mode will cancel all outstanding ISAM locks (locks created by **LockData** will NOT be released). RESET is used when your program is traversing the data in key sequential order, and after each update, the previous locks should be released, and new locks enabled for the next transaction.

The READREC mode should be used to signal that the record is under review by one or more readers, and that no update should be performed. If records are read without read locks (or write locks), then another process may update the record while it is under review by other processes. Usually, this latter situation is acceptable.

NOTE: many systems do not support read locks. If read locks are not supported, c-tree Plus returns a no-error condition (i.e., the read lock is not attempted, but no error is reported). The FairCom Server *DOES* support read locks and write locks.

To avoid unnecessary data record locks in the middle of a transaction, there are four **LockISAM** modes which switch the locking off and on again: SUSPEND, RESTORE, RESTRED and RESTORE_BLK. Use the SUSPEND mode when it is desired to scan one or more data records which WILL NOT BE UPDATED (or for which READ LOCKS ARE NOT REQUIRED) in the middle of a transaction which has already enabled data record locks. After scanning these records, and before returning to your update activities, call **LockISAM** with the RESTORE mode to cause the automatic data record write locks to be enabled; or call **LockISAM** with the RESTRED mode to cause automatic data record read locks to be enabled; or call **LockISAM** with the RESTORE_BLK mode to restore ENABLE_BLK.

For example, suppose your program is performing updates to customer invoices. The updates involve price extension computations. To perform the computations, your program must search a separate ISAM file which maintains the prices. Since your program will not modify the price records, you call **LockISAM(SUSPEND)** before searching the price records, and **LockSAM(RESTORE)** before continuing with the invoice updates. See the example below.

If you are using Transaction functions you should refer to Chapter PRG 11 for information on how transaction processing interacts with the LockISAM function.

RETURN Errors can only occur in the RESET, ENABLE, or ENABLE_BLK modes.

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful operation.
112	IPND_ERR	A call to ENABLE locks found some record locks still pending. This error check is made to help ensure that your program logic has not started a new transaction before clearing the locks associated with an old transaction. However, it will also cause an error if you acquire some locks and then issue a redundant call to **LockISAM** in the ENABLE mode.
113	INOL_ERR	No more room left in the internal lock table maintained by **LockISAM**. Increase the MAX_LOCKS parameter in *ctoptn.h*. If a lock is denied by the operating system because of an existing lock, then other ISAM routines will return an error 42 (DLOK_ERR).

EXAMPLE

```
#include "ctoptn.h"
#include "ctaerr.h"
#include "ctdecl.h"

COUNT  inv_idx,inv_fil,price_idx,price_fil;
TEXT target[9];

struct invd  {
   TEXT   delflg;
   TEXT   invn[8];
   TEXT   dscrp[60];
   TEXT   part[12];
   float  quant;
   float  price;
   float  exten;
} cur_info;

struct {
   TEXT   pdlflg;
   TEXT   ppart[12];
   float  pprice;
} price_rec;

main() {
   printf("\nEnter invoice number: ");
   scanf("%8s",target);
   if (LockISAM(ENABLE) || FirstInSet(inv_idx,target,
         &cur_info,8)) {
     printf("\n\nError starting SCAN with codes %d %d",
         isam_err,isam_fil);
```

```
            LockISAM(FREE);
            return;
          }
          while (!isam_err) {
            LockISAM(SUSPEND); /* turn off lock acquisition */
            get_price_extension();
            LockISAM(RESTORE); /* restore lock acquisition */
            ReWriteRecord(inv_fil,&cur_info);
            ResetRecord(inv_fil,SWTCURI);
                                /* reset current ISAM record */
            LockISAM(RESET);
            NextInSet(inv_idx,&cur_info);
          }
          LockISAM(FREE);
        }

        VOID get_price_extension()
        {
          if (GetKey(price_idx,cur_info.part,&price_rec)) {
            printf("\nCould not find part# %s.",cur_info.part);
            return;
          }
          new_info.exten = (cur_info.price = price_rec.pprice) *
            cur_info.quant;
        }
```

LIMITATIONS

The FairCom Server supports both read and write locks on records. If you are not using the server for multi-user applications, be aware that the read locks may show success even if read locks are not implemented.

NewData

Get next available fixed length data record position.

SHORT NAME NEWREC

TYPE Low level data file function

DECLARATION

```
LONG NewData(datno);
COUNT datno;
```

DESCRIPTION **NewData** determines the next available data record position for fixed length data file number *datno*. If records have been deleted (via **ReleaseData**), then they are reused by **NewData** before the file size is extended. Each call to **NewData** increments the serial number associated with the file. (The ISAM level key segment mode of three (3) permits this serial number to be part of a key value to enable chronologically, or reverse chronologically, ordered key values. See Chapter PRG 5)

NOTE: In multi-user systems, **NewData** automatically acquires a data record lock on the newly acquired record. This lock should be released by the **LockData** function after you have written the contents of the new record.

Each data file starts with a header record which contains information about its size, record length and deleted records. The header record begins at byte position zero. The first usable data record position is given by the expression:

```
((128 + record_length - 1) / record_length) * record_length
```

where we assume the integer division truncates the result. For example, a record length of 30 bytes means that the first actual data record will start at byte position 150. The above expression guarantees that the first usable record always starts after the first 128 bytes of the file AND on a record length boundary.

NOTE: files that are members of Superfiles do not satisfy the above expression, nor do they fall on record length boundaries.

RETURN **NewData** returns the data record position of the next available record. If an error occurs, **NewData** returns a zero and uerr_cod is set to a non-zero value.

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful operation.

Value	Symbolic Constant	Explanation
31	DELFLG_ERR	First byte of record about to be reused does not equal the delete flag (ff hex). Either file is corrupt and must be rebuilt, or the C255 constant defined in *ctcmpl.h* is not properly set.
48	FMOD_ERR	*datno* is assigned to a variable length file.

EXAMPLE

```
LONG recd;
COUNT   dfil;
TEXT   *recptr;

if ((recd = NewData(dfil)) &&
      WriteData(dfil,recd,recptr) == 0 &&
      LockData(dfil,FREE,recd) == 0)
   printf("\nSuccessful addition of new record.");
```

SEE ALSO **ReleaseData** and **LockData**.

NewVData

Get next available variable length data record position.

SHORT NAME NEWVREC

TYPE Low level data file function

DECLARATION

```
LONG NewVData(datno,varlen);
COUNT  datno;
VRLEN varlen;
```

DESCRIPTION **NewVData** determines the next available data record position for variable length data file number *datno*. If records have been deleted (via **ReleaseVData**), and if they are large enough, then they are reused by **NewVData** before the file size is extended. Each call to **NewVData** causes the serial number associated with the file to be incremented. The ISAM level key segment mode of three (3) permits this serial number to be part of a key value to enable chronologically, or reverse chronologically, ordered key values. See Chapter PRG 5.

NOTE: In multi-user systems, NewVData automatically acquires a data record lock on the newly acquired record. This lock should be released by the LockData function after you have written the contents of the new record.

Each variable length record is preceded by a ten byte control block which specifies whether the record is active or not, how long the record is, and how much of the record is actually in use.

RETURN NewVData returns the data record position of the next available record. If an error occurs, **NewVData** returns a zero and uerr_cod is set to a non-zero value.

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful operation.
48	FMOD_ERR	*datno* is assigned to a fixed length file.
147	VDLFLG_ERR	The record to be reused is not marked deleted. The file is corrupt and must be rebuilt.

EXAMPLE

```
LONG   recd;
COUNT  dfil;
VRLEN  varlen;
TEXT   *recptr;

varlen = strlen(recptr);
```

```
if ((recd = NewVData(dfil)) == 0L ||
WriteVData(dfil,recd,recptr,varlen) ||
      LockData(dfil,FREE,recd))
   printf("\nUnsuccessful addition of new record.");
```

SEE ALSO **ReleaseVData** and **LockData**.

NextInSet

Read the next data record in the current key value set.

SHORT NAME NXTSET

TYPE ISAM function

DECLARATION

```
COUNT NextInSet(keyno,recptr)
COUNT keyno;
VOID *recptr;
```

DESCRIPTION

NextInSet reads the next data record in the set of records whose key values match the target key established in the most recent call to **FirstInSet**. If successful, the next record becomes the current ISAM record for the associated data file. If an error occurs, or no next record exists in the set, the current ISAM record remains the same, and the current key value set becomes undefined.

If the associated data file has variable length records, then only the minimum record length portion (defined by *datlen* in the original call to CreateDataFile) is actually read into the buffer pointed to by *recptr*. If you wish to read the entire variable length record into the same or a different buffer, issue a call to **ReReadVRecord** after the call to **NextInSet**. Note that **ReReadVRecord** requires you to specify the size of the buffer area so that it can check if sufficient space is available.

Notice that you do not directly identify the data file number involved. The ISAM parameter file described in Chapter PRG 5 contains the correspondence between the index file number (*keyno*) and the associated data file.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful update of current ISAM record.
33	DNUL_ERR	*recptr* is NULL. No data file read performed.
42	DLOK_ERR	Could not get lock on data record. No data file read performed.
100	ICUR_ERR	No current ISAM record.
101	INOT_ERR	No more records are in the target set established by the call to **FirstInSet**.
118	SKEY_ERR	Either **FirstInSet** has not been called or the last call to **FirstInSet** was not for index number *keyno*.

EXAMPLE

```
COUNT          acct_idx;
TEXT           cust_acct[10],recbuf[320];

scanf("%6s",cust_acct);
if (FirstInSet(acct_idx,cust_acct,recbuf,6) ==
          NO_ERROR)
   ReReadVRecord(inv_dat,recbuf,320);
while (isam_err == NO_ERROR) {
   post_interest(recbuf);
   if (NextInSet(acct_idx,recbuf) == NO_ERROR)
     ReReadVRecord(inv_dat,recbuf,320);
}
```

LIMITATIONS

No check is made to determine if *recptr* points to a region sufficiently large to accept a data record. If the area is too small, either code or data will be clobbered.

SEE ALSO

FirstInSet, **LastInSet**, **PreviousInSet** and **ChangeSet**

NextKey

Find the next entry in an index file.

SHORT NAME NXTKEY

TYPE Low level index file function

DECLARATION

```
LONG NextKey(keyno,idxval);
COUNT keyno;
VOID *idxval;
```

DESCRIPTION **NextKey** moves to the next key sequential entry in the index file *keyno*. If successful, it copies the index entry into the space pointed to by *idxval*. **NextKey** will fail if it is the first index search routine called after the index is opened, or if the entry found before the call to **NextKey** is in fact the last entry in the index.

Repeated calls to **NextKey** move you through the index in ascending key value order.

RETURN **NextKey** returns the data record position associated with the next entry in the index file. If **NextKey** fails or an error occurs, then **NextKey** returns a zero. After a call to **NextKey** which returns a zero, check the value of uerr_cod: if uerr_cod is non-zero, an error condition was detected; otherwise no more key values exist.

EXAMPLE

```
LONG        recbyt;
TEXT        idxval[10];
COUNT       keyno;

recbyt = FirstKey(keyno,idxval);
while (recbyt) {
   printf("\nIndex Entry %.10s is found in record #%ld",
      idxval,recbyt);
   recbyt = NextKey(keyno,idxval);
}
```

LIMITATIONS No check is made to determine if *idxval* points to a region sufficiently large to accept a key value from the index file. If the area is too small, either code or data will be clobbered.

SEE ALSO **GetGTKey** and **PreviousKey**

NextRecord

Read the next data record.

SHORT NAME NXTREC

TYPE ISAM function

DECLARATION

```
COUNT NextRecord(filno,recptr)
COUNT filno;
VOID *recptr;
```

DESCRIPTION

If **NextRecord** is called with a *filno* corresponding to an index, then **NextRecord** reads the next data record, based on key sequential order for index number *filno*. If it is called with a *filno* corresponding to a data file, then **NextRecord** reads the next record in physical sequential order, automatically skipping over deleted records. If successful, the next record becomes the current ISAM record for the associated data file. If an error occurs, or no next record exists, the current ISAM record remains the same.

If the *filno* is a data file that is a member of a Superfile, NextRecord cannot be guaranteed to give you the next physical record.

If the associated data file has variable length records, then only the minimum record length portion (defined by *datlen* in the original call to **CreateDataFile**) is actually read into the buffer pointed to by *recptr*. If you wish to read the entire variable length record into the same or a different buffer, issue a call to **ReReadVRecord** after the call to **NextRecord**. Note that **ReReadVRecord** requires you to specify the size of the buffer area so that it can check if sufficient space is available.

When called with an index number, you do not directly identify the data file number involved. The ISAM parameters described in Chapter PRG 5 contain the correspondence between the index number (*filno*) and the associated data file.

You can use **NextRecord** to move sequentially through a data file based on one index and then switch to another index. For example, you might move through an inventory file in part number order and then switch to part name order by changing the *keyno* parameter in the **NextRecord** call.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful update of current ISAM record.
33	DNUL_ERR	*recptr* is NULL. No data file read performed.

Value	Symbolic Constant	Explanation
42	DLOK_ERR	Could not get lock on data record. No data file read performed.
100	ICUR_ERR	No current ISAM record.
101	INOT_ERR	**NextRecord** not found. Either current ISAM record is the last record or its key value for index file *keyno* is missing.

EXAMPLE

```
COUNT acct_idx,test;
TEXT  recbuf[320];

test = FirstRecord(acct_idx,recbuf);
while (test == NO_ERROR) {
   post_interest(recbuf);
   test = NextRecord(acct_idx,recbuf);
}
```

LIMITATIONS

No check is made to determine if *recptr* points to a region sufficiently large to accept a data record. If the area is too small, either code or data will be clobbered.

SEE ALSO

PreviousRecord and **ReReadVRecord**

NbrOfKeyEntries

Number of entries in index file.

SHORT NAME IDXENT

TYPE Low level index file function

DECLARATION

```
LONG NbrofKeyEntries(keyno)
COUNT keyno;
```

DESCRIPTION **NbrOfKeyEntries** returns the number of entries in index file number *keyno*.

RETURN If an error occurs or the index file is empty, **NbrOfKeyEntries** returns a value of zero. Otherwise, **NbrOfKeyEntries** returns the number of entries in the index file. After a call to **NbrOfKeyEntries** which returns a zero, check the value of uerr_cod: if uerr_cod is non-zero, an error condition was detected; otherwise no key values are in the index.

EXAMPLE

```
COUNT    keyno;

scanf("%d",&keyno);
printf("\nThere are %ld entries in index #%d.",
NbrofKeyEntries(keyno),keyno);
```

LIMITATIONS In non-server multi-user systems or when updates are immediately forced to disk (i.e., the disk I/O protocol is set to FPUTFGET), the time to add or delete an index entry can be reduced if the number of index entries is not maintained. As shipped, c-tree Plus does maintain this information, but the c-tree Plus Installation Guide shows how to disable the entry count. Therefore, we suggest you only use this function if it is very important to quickly determine the number of index entries.

NbrOfRecords

Get the number of active records in fixed length data file.

SHORT NAME

DATENT

TYPE

Low level fixed length data file function

DECLARATION

```
LONG NbrOfRecords(datno)
COUNT datno;
```

DESCRIPTION

NbrOfRecords returns the number of active data records for the fixed length data file designated by *datno*.

Each time a new record is added to the file, the count of active records is incremented. Each time a record is deleted, the active count is decremented. A record is added by the low level function **NewData** or the ISAM function **AddResource**. A record is deleted by the low level function **ReleaseData** or the ISAM function **DeleteRecord**.

RETURN

If an error occurs or if there are no active data records, **NbrOfRecords** returns a value of zero. Otherwise, **NbrOfRecords** returns the number of active fixed data records in the data file. After a call to **NbrOfRecords** which returns a zero, check the value of uerr_cod: if uerr_cod is non-zero, an error condition was detected; otherwise no active records are in the file.

EXAMPLE

```
COUNT   datno;

scanf("%d",&datno);
printf("\nThere are %ld active records in file #%d.",
   NbrOfRecords(datno),datno);
```

OpenFile

Open Data or Index file.

SHORT NAME OPNFIL

TYPE Low level file function

DECLARATION

```
COUNT openFile(filno,filnam,filmod)
COUNT filno,filmod;
TEXT *filnam;
```

DESCRIPTION

OpenFile opens the file with name pointed to by *filnam* in the mode specified by *filmod*. All subsequent references to the file are made via the *filno* value. *filno* must be in the range 0 to *fils*-1 where *fils* is the second parameter in the initialization routine **InitCTree**.

If *filnam* references an index file with additional index members (i.e., more than one index is stored in the physical file), **OpenFile** opens all indices included in the file. The *filno* assigned to an additional index member is equal to the *filno* in the call to **OpenFile** plus the member number of the additional index.

Before a file can be opened, it must be created with either **CreateDataFile** or **CreateIndexFile**.

At the time a file is opened, you cannot change from a fixed length to a variable length file. (When **CreateDataFile** is called, the *filmod* value determines whether a data file has fixed or variable length records.)

Some values must be used exclusively of others. For instance, you cannot use both SHARED and EXCLUSIVE. You cannot use more than one mode from any line in the listing below. In addition, you **must** use one of the modes from each of the first two groups.

```
PERMANENT          VIRTUAL
EXCLUSIVE          SHARED     READFIL
CHECKLOCK
DUPCHANEL
WRITETHRU
```

NOTE: file modes CHECKLOCK and DUPCHANEL are available only with the FairCom Server.

SUPERFILES: An index file can be a member of a Superfile, as discussed in Chapter PRG 8. The name of the member must be formed as follows:

```
<name of superfile>!<name of member>
```

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful open.
12	FNOP_ERR	Could not open file. Most likely cause is that file does not exist. Check that *filnam* points to correct name. Also, FNOP_ERR will be returned in a multi-user system if the file has been locked by another process.
13	FUNK_ERR	Type of file is unknown. Header record is clobbered.
14	FCRP_ERR	File found corrupt at open.
20	KMIN_ERR	Key length too large for node size.
22	FNUM_ERR	File number is out of range.
40	KSIZ_ERR	At time index was created, *sect* parameter of **InitCTree** was larger.
43	FVER_ERR	Current c-tree Plus configuration (specified in *ctoptn.h*) is incompatible with file.
45	KLEN_ERR	Key length exceeds MAXLEN in *ctoptn.h*.
46	FUSE_ERR	*filno* is already in use.

EXAMPLE

```
TEXT  filnam[15];
COUNT invfil;
scanf("%d %14s",invfil,filnam);
if (OpenFile(invfil,filnam,(SHARED | PERMANENT)))
   printf("\nUnsuccessful file open.");
```

LIMITATIONS

In non-sever multi-user implementations of c-tree Plus, the locking routines depend on the operating system and possibly the compiler. In this case you may have to code your own locking routines. The files *ctclib.c* provided in each sub-directory contain locking routines and shared file opening facilities for the popular multi-user and network systems.

SEE ALSO

InitCTree, **CreateDataFile**, **CreateIndexFile**, **OpenFileXtd**.

OpenFileWithResource

Incremental ISAM open, based on the IFILE Resource.

SHORT NAME OPNRFIL

TYPE ISAM function

DECLARATION

```
COUNT OpenFileWithResource(filno,filnam,filmod)
COUNT filno,filmod;
TEXT *filnam;
```

DESCRIPTION

OpenFileWithResource opens the data file *filnam* and (optional) index file referenced by the IFIL resource information stored in that file. This resource was automatically added to the file when the file was created with **CreateIFile** unless Resources have been disabled for this file. All subsequent references to the file are made via the *filno* value. *filno* must be in the range 0 to *fils*-1 where *fils* is the second parameter in the initialization routine **InitCTree**.

When using an incremental ISAM file, all indices associated with the data file must be contained in a single, companion index.

Consecutive file numbers are assigned to the data file and any (optional) indices associated with the data file. If *filno* is less than zero, then c-tree Plus finds the first available block of consecutive file numbers which can accommodate the data file and its indices. If such a block of numbers is found, then the first number in the block is assigned to the data file and will be the value returned by the function. If *filno* is greater than or equal to zero, then *filno* is assigned to the data file and returned from the function.

RETURN

The value returned by **OpenFileWithResource** is the file number assigned to the data file. If an error occurs, the function returns a -1, and the error code can be found in the variable *isam_err*. The error codes are:

Value	Symbolic Constant	Explanation
12	FNOP_ERR	Could not open file(s). They do not exist, or there is a lock conflict.
14	FCRP_ERR	File appears corrupt at open: rebuild.
20	KMIN_ERR	Key length too large for node size.
22	FNUM_ERR	File number is out of range.
45	KLEN_ERR	Key length exceeds MAXLEN parameter in *ctoptn.h*.

Value	Symbolic Constant	Explanation
46	FUSE_ERR	If *filno* > = 0, then file number (range) already in use. If *dfilno* < 0, then a block of consecutive numbers large enough to handle the data file and it's associated indices could not be found.
47	FINT_ERR	c-tree Plus has not been initialized. Call **InitISAM** before the first call to **OpenFileWithResource**.
107	IDRK_ERR	Too many index files for one data file. Increase MAX_DAT_KEY parameter in *ctoptn.h*.
109	IKRS_ERR	Too many key segments. Increase MAX_KEY_SEG parameter in *ctoptn.h*.

EXAMPLE

```
TEXT  filnam[15];
COUNT invfil;
scanf("%d %14s",invfil,filnam);
if (OpenFileWithResource(invfil,filnam,(SHARED |
PERMANENT))< 0)
   printf("\nUnsuccessful file open.");
```

LIMITATIONS

All indices must be in a single index file. In addition, the data file could not be created with resources disabled.

SEE ALSO

OpenIFile, **InitISAM**, **CreateIFile**, **CloseISAM** **OpenFileWithResourceXtd**, and Chapter PRG 5 which describes the ISAM parameters.

OpenFileWithResourceXtd

Incremental ISAM open, based on the IFILE Resource (extended).

SHORT NAME OPNRFILX

TYPE Extended ISAM function

DECLARATION

```
COUNT
OpenFileWithResourceXtd(filno,filnam,filmod,fileword)
COUNT filno,filmod;
TEXT *filnam;
TEXT *fileword;
```

DESCRIPTION

OpenFileWithResourceXtd is a variation of **OpenFileWithResource** that permits the use of the FairCom Server's security system. Please read the description of **OpenFileWithResource** before proceeding with this description, as we will only describe the extended features here.

fileword is the **file password**, as explained on page PRG 12-4. If no file password is required for this file the *fileword* should be null.

RETURN

The following error codes may be seen in addition to those shown for **OpenFileWithResource**.

Value	Symbolic Constant	Explanation
456	SACS_ERR	You are not a member of a group that can access this file.
457	SPWD_ERR	Invalid password.

OpenFileXtd

Extended Open Data or Index file.

SHORT NAME OPNFILX

TYPE Extended low level file function

DECLARATION

```
COUNT OpenFileXtd(filno,filnam,filmod,fileword)
COUNT filno,filmod;
TEXT *filnam;
TEXT *fileword;
```

DESCRIPTION

OpenFileXtd is a variation of **OpenFile** that permits the use of the FairCom Server's security system. Please read the description of **OpenFile** before proceeding with this description, as we will only describe the extended features here.

fileword is the **file password**, as explained on page PRG 12-4. If no file password is required for this file the *fileword* should be null.

RETURN

The following error codes may be seen in addition to those shown for **OpenFile**.

Value	Symbolic Constant	Explanation
456	SACS_ERR	You are not a member of a group that can access this file.
457	SPWD_ERR	Invalid password.

OpenIFile

Incremental ISAM open.

SHORT NAME OPNIFIL

TYPE ISAM function

DECLARATION

```
COUNT OpenIFile(ifilptr)
IFIL *ifilptr;
```

DESCRIPTION

OpenIFile opens the data file and (optional) index file referenced by the IFIL structure pointed to by *ifilptr*. When using an incremental ISAM file, all indices associated with the data file must be contained in a single, companion index. The file name pointed to by the IFIL structure (see ifilptr->pfilnam in Chapter PRG 5) SHOULD NOT include an extent: ".dat" and ".idx" are automatically appended to the file name for the data file and index file, respectively.

If you desire the files to be opened in directories that are not specified until run-time, then the file name pointed to by the IFIL structure should not be finalized until run-time.

Consecutive file numbers are assigned to the data file and any (optional) indices associated with the data file. The IFIL structure contains two file numbers: a permanent file number (dfilno), and a temporary file number (tfilno). If dfilno is less than zero, then c-tree Plus finds the first available block of consecutive file numbers which can accommodate the data file and its indices. If such a block of numbers is found, then the first number in the block is assigned to the data file and to tfilno. If dfilno is greater than or equal to zero, then dfilno is assigned to the data file and to tfilno.

See Chapter PRG 5 for a complete description of the IFIL structure.

RETURN

Upon successful open, ifilptr->tfilno contains the file number assigned to the data file; and the associated indices are assigned consecutive key numbers starting one (1) greater than the data file number.

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful open of ISAM files.
12	FNOP_ERR	Could not open file(s).
14	FCRP_ERR	File appears corrupt at open: rebuild.
20	KMIN_ERR	Key length too large for node size.

Value	Symbolic Constant	Explanation
22	FNUM_ERR	File number is out of range.
45	KLEN_ERR	Key length exceeds MAXLEN parameter in *ctoptn.h*.
46	FUSE_ERR	If *ifilptr->dfilno* >= 0, then file number (range) already in use. If *ifilptr->dfilno* < 0, then a block of consecutive numbers large enough to handle the data file and it's associated indices could not be found.
47	FINT_ERR	c-tree Plus has not been initialized. Call **InitISAM** before the first call to **OpenIFile**.
107	IDRK_ERR	Too many index files for one data file. Increase MAX_DAT_KEY parameter in *ctoptn.h*.
109	IKRS_ERR	Too many key segments. Increase MAX_KEY_SEG parameter in *ctoptn.h*.
115	ISLN_ERR	Key segments do not equal key length.

EXAMPLE

```
extern IFIL  customer;
TEXT         full_name[64];
extern TEXT *path_name,*file_name;
extern COUNT data_no;

strcpy(full_name,path_name);
strcat(full_name,file_name);
customer->pfilnam = full_name;
if (OpenIFile(&customer))
   printf("\nCould not open %s
(%d,%d)\n",isam_err,isam_fil);
else
   data_no = customer->tfilno;
```

LIMITATIONS

The data file and index files must be numbered consecutively, and all indices must be in a single index file.

SEE ALSO

OpenIFileXtd, **InitISAM**, **CreateIFile**, **CloseISAM** and Chapter PRG 5 which describes the ISAM parameters.

OpenIFileXtd

Extended Incremental ISAM open.

SHORT NAME OPNIFILX

TYPE Extended ISAM function

DECLARATION

```
COUNT OpenIFileXtd(ifilptr,dataextn,indxextn,fileword)
IFIL *ifilptr;
TEXT  *dataextn;
TEXT  *indxextn;
TEXT  *fileword;
```

DESCRIPTION

OpenIFileXtd is a variation of **OpenIFile** that permits the use of the FairCom Server's security system. In addition, you can use it to change the file name extensions from the defaults that are used in **OpenIFile**. Please read the description of **OpenIFile** before proceeding with this description, as we will only describe the extended features here.

fileword is a **file password**, as explained on page PRG 12-4. If there is no password for this file then set *fileword* to a null.

dataextn and *indxextn* are pointers to buffers that can be used to specify the data and index file name extensions, respectively. The extensions are 8 byte ASCIIZ strings. If they are null pointers then the default extension will be used, ".dat" for data files and ".idx" for index files. If you want no extension then you should pass a pointer to a buffer that contains only blanks terminated by a null character.

RETURN

The following error codes may be seen in addition to those shown for **OpenIFile-Xtd**.

Value	Symbolic Constant	Explanation
456	SACS_ERR	You are not a member of a group that can access this file.
457	SPWD_ERR	Invalid password.

OpenISAM

Open all the ISAM files.

SHORT NAME OPNISAM

TYPE ISAM function

DECLARATION

```
COUNT OpenISAM(filnam)
TEXT *filnam;
```

DESCRIPTION

OpenISAM opens the parameter file whose name is pointed to by *filnam* and opens the data and index files specified in the parameter file. Through the use of virtual files, the parameter file may contain more files than available file descriptors. c-tree Plus will automatically manage the opening and closing of the virtual files. Application programs using the ISAM level functions must make a call to **CreateISAM**, **InitISAM** or **OpenISAM** before any other c-tree Plus functions are used. If any one of these three functions has been called, then **CloseISAM** must be called before anyone of them can be invoked again. This implies that no more than one ISAM parameter file may be active at one time. However, the incremental ISAM functions allow for more flexibility in file manipulations.

In order to use incremental ISAM opens and creates along with a parameter file, you must increase the value of the second parameter in the parameter file (idxs) by the number of data files and indices to be utilized with the incremental routines. For example, if the incremental routines would be opening two additional data files, each with five indices, then the idx parameter in your ISAM parameter file should be increased by 12.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful open or create of ISAM files.
10	SPAC_ERR	Not enough dynamic memory.
11	SPRM_ERR	Illegal **InitCTree** parameters in first record of parameter file.
12	FNOP_ERR	Could not open file.
13	FUNK_ERR	Unknown file type.
14	FCRP_ERR	File found corrupt at open.
20	KMIN_ERR	Key length too large for node size.
21	DREC_ERR	Record length too small.

Value	Symbolic Constant	Explanation
43	FVER_ERR	Current c-tree Plus configuration (specified in *ctoptn.h*) is incompatible with existing file.
45	KLEN_ERR	Key length exceeds MAXLEN parameter in *ctoptn.h*.
102	INOD_ERR	Could not open ISAM parameter file.
103	IGIN_ERR	Could not read first record of ISAM parameter file.
104	IFIL_ERR	Too many files specified. Increase MAXFIL in *ctoptn.h*.
106	IDRI_ERR	Could not read data file description record in parameter file.
107	IDRK_ERR	Too many index files for one data file. Increase MAX_DAT_KEY parameter in *ctoptn.h*.
108	IMKY_ERR	*keyno* for an index member is out of sequence. *keyno* must equal the host index *keyno* plus the member number.
109	IKRS_ERR	Too many key segments. Increase MAX_KEY_SEG parameter in *ctoptn.h*.
110	ISRC_ERR	Could not read key segment description record in parameter file.
111	IKRI_ERR	Could not read index file description record in parameter file.
115	ISLN_ERR	Key segments do not equal key length.
117	IMRI_ERR	Could not read additional index member record.
125	IINT_ERR	c-tree Plus already initialized. Call **CloseISAM** before recalling **OpenISAM**.
182	IKSR_ERR	Not enough c-tree Plus file control blocks for the r-tree temporary files. Increase MAXFIL in *ctoptn.h*.

EXAMPLE

```
TEXT        isam_par[64];

printf("\nEnter ISAM parameter file name: ");
scanf("%63s",isam_par);
if (OpenISAM(isam_par))
   printf("\n%s files could not be opened. (%d  %d)",
     isam_err,isam_fil);
```

LIMITATIONS

It is not possible to open some of the files specified in the parameter file and create others. They must all be opened by **OpenISAM**. However, the incremental ISAM

open (**OpenIFile**) and create (**CreateIFile**) functions provide more flexibility. If you are adding a new index file to an established set of ISAM files, you should

- update the parameter file to include the new index; and
- use the rebuild utility to create and build the new index file (respond no when asked to force rebuild).

SEE ALSO

CloseISAM, **InitISAM**, **CreateISAM**, **OpenIFile**, **OpenISAMXtd**, **CreateIFile** and Chapter PRG 5 which describes the parameter file contents.

OpenISAMXtd

Open all the ISAM files (extended version).

SHORT NAME OPNISAMX

TYPE Extended ISAM function

DECLARATION

```
COUNT
OpenISAMXtd(filnam,dbufs,userprof,userid,userword,
                    servname,fileword)
TEXT *filnam;
COUNT dbufs;
COUNT userprof;
TEXT *userid;
TEXT *userword;
TEXT *servname;
TEXT  *fileword;
```

OpenISAMXtd is a variation of **OpenISAM** that permits the use of the FairCom Server's security system, as well as setting the user profile mask. Please read the description of **OpenISAM** before proceeding with this description, as we will only describe the extended features here.

dbufs specifies the number of data buffers.

userprof is the **user profile mask**. If you set this to the value USERPRF_NTKEY you will **disable** the **automatic TransformKey** feature. See the function description for **TransformKey** for more information. If you do not wish to disable this feature, set *userprof* to zero.

userid is a pointer to a buffer containing your user ID. See page PRG 12-2 for more information on user ID's.

userword is a pointer to a buffer containing your user password. See page PRG 12-2 for more information on user passwords.

servname is a pointer to a server name if you are going to use a server other than the default server, in a multi-server environment. See the Server documentation for more information.

fileword is the **file password**, as explained on page PRG 12-4. If there is no file password for this file, *fileword* should be a null.

RETURN The following error codes may be seen in addition to those shown for **OpenISAMXtd**.

Value	Symbolic Constant	Explanation
450	LUID_ERR	Invalid user ID.
451	LPWD_ERR	Invalid user password.
453	NSRV_ERR	Invalid server name.
456	SACS_ERR	You are not a member of a group that can access this file.
457	SPWD_ERR	Invalid password.

PositionSet

Establish current ISAM record as current position in set.

SHORT NAME MIDSET

TYPE ISAM function

DECLARATION

```
COUNT PositionSet(keyno,recptr,siglen)
COUNT keyno;
VOID *recptr;
COUNT siglen;
```

DESCRIPTION

PositionSet extracts the key value for index *keyno* from the current ISAM record, and defines a set based on the first *siglen* bytes of this key value. This permits an application to "jump" into the middle of a set using the current ISAM record to define the set. A successful **PositionSet** defines a current key value set, and subsequent calls to **NextInSet** and **PreviousInSet** will read the other records in the set (i.e., those records whose keys also match the first *siglen* bytes of *target*). If an error occurs or no key value matches the target, no key value set is defined.

Notice that you do not directly identify the data file number involved. The ISAM parameter file described in Chapter PRG 5 contains the correspondence between the index file number (*keyno*) and the associated data file.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful update of current ISAM record and definition of key value set.
33	DNUL_ERR	*recptr* is NULL. No data file read performed.
42	DLOK_ERR	Could not get lock on data record. No data file read performed.
100	ICUR_ERR	No current ISAM record.
101	INOT_ERR	No matching key value found.
119	SKTY_ERR	*keyno* must reference a left-to-right scan key type.

EXAMPLE

```
COUNT  keyfil;
TEXT target[24],recbuf[320];

printf("\nEnter set value:");
```

```
scanf("%23s",target);
LockISAM(ENABLE);
FirstInSet(keyfil,target,recbuf,strlen(target));
while (isam_err == NO_ERROR) {
   process_data();
   LockISAM(RESET);
   while (NextInSet(keyfil,recbuf) == DLOK_ERR)) {
     LockISAM(SUSPEND);
     PositionSet(keyfil,recbuf,strlen(target));
     LockISAM(RESTORE);
   }
}
```

LIMITATIONS

PositionSet can only be called for keys which are scanned from left to right. *keytyp*'s of 1, 2 and 3 are not permitted.

No check is made to determine if *recptr* points to a region sufficiently large to accept a data record. If the area is too small, either code or data will be clobbered.

SEE ALSO

NextInSet, **PreviousInSet**, **FirstInSet**, **LastInSet** and **ChangeSet**

PreviousInSet

Read the previous data record in the current key value set.

SHORT NAME PRVSET

TYPE ISAM function

DECLARATION

```
COUNT PreviousInSet(keyno,recptr)
COUNT keyno;
VOID *recptr;
```

DESCRIPTION

PreviousInSet reads the previous data record in the set of records whose key values match the target key established in the most recent call to **FirstInSet**. If successful, the previous record becomes the current ISAM record for the associated data file. If an error occurs, or no previous record exists in the set, the current ISAM record remains the same, and the current key value set becomes undefined.

If the associated data file has variable length records, then only the minimum record length portion (defined by *datlen* in the original call to **CreateDataFile**) is actually read into the buffer pointed to by *recptr*. If you wish to read the entire variable length record into the same or a different buffer, issue a call to **ReReadVRecord** after the call to **PreviousInSet**. Note that **ReReadVRecord** requires you to specify the size of the buffer area so that it can check if sufficient space is available.

Notice that you do not directly identify the data file number involved. The ISAM parameter file described in Chapter PRG 5 contains the correspondence between the index file number (*keyno*) and the associated data file.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful update of current ISAM record.
33	DNUL_ERR	*recptr* is NULL. No data file read performed.
42	DLOK_ERR	Could not get lock on data record. No data file read performed.
100	ICUR_ERR	No current ISAM record.
101	INOT_ERR	No more records are in the target set established by the call to **FirstInSet**.
118	SKEY_ERR	Either **FirstInSet** has not been called or the last call to **FirstInSet** was not for index number *keyno*.

EXAMPLE

```
COUNT          acct_idx;
TEXT           recbuf[320];

if (PreviousInSet(acct_idx,recbuf) == INOT_ERR)
   printf("\nwent back past beginning of set\n");
else if (isam_err == SKEY_ERR)
   printf("\nno current key value set for index #%d",
     acct_idx);
```

LIMITATIONS

No check is made to determine if *recptr* points to a region sufficiently large to accept a data record. If the area is too small, either code or data will be clobbered.

SEE ALSO

FirstInSet, LastInSet, NextInSet, ChangeSet and **PreviousRecord**

PreviousKey

Find the previous entry in an index file.

SHORT NAME PRVKEY

TYPE Low level index file function

DECLARATION

```
LONG PreviousKey(keyno,idxval);
COUNT keyno;
VOID *idxval;
```

DESCRIPTION **PreviousKey** moves backward to the preceding entry in the index file *keyno*. If successful, it copies this index entry into the space pointed to by *idxval*. **PreviousKey** will fail if it is the first index search routine called after the index is opened, or if the entry found before the call to **PreviousKey** is in fact the first entry in the index.

Repeated calls to **PreviousKey** move you through the index in descending key value order.

RETURN **PreviousKey** returns the data record position associated with the preceding entry in the index file. If **PreviousKey** fails or an error occurs, then **PreviousKey** returns a zero. After a call to **PreviousKey** which returns a zero, check the value of uerr_cod: if uerr_cod is nonzero, an error condition was detected; otherwise no more key values exist.

EXAMPLE

```
LONG recbyt;
TEXT idxval[10];
COUNT  keyno;

recbyt = LastKey(keyno,idxval);
while (recbyt) {
   printf("\nIndex Entry %.10s is found in record #%ld",
     idxval,recbyt);
   recbyt = PreviousKey(keyno,idxval);
}
```

LIMITATIONS No check is made to determine if *idxval* points to a region sufficiently large to accept a key value from the index file. If the area is too small, either code or data will be clobbered.

SEE ALSO **GetLTKey** and **NextKey**

PreviousRecord

Read the preceding data record.

SHORT NAME PRVREC

TYPE ISAM function

DECLARATION

```
COUNT PreviousRecord(filno,recptr)
COUNT filno;
VOID *recptr;
```

DESCRIPTION

If **PreviousRecord** is called with a *filno* corresponding to an index, then **PreviousRecord** reads the previous data record, based on key sequential order for index number *filno*. If it is called with a *filno* corresponding to a data file, then **PreviousRecord** reads the previous record in physical sequential order, automatically skipping over deleted records. If successful, the previous record becomes the current ISAM record for the associated data file. If an error occurs, or no previous record exists, the current ISAM record remains the same.

NOTE: *filno* cannot be a data file number if any of the following are true:

- the data file is set up for variable length records;
- the data file is a member of a Superfile;
- the data file has Resources enabled (the default state).

In any of the above cases the error FMOD_ERR will be returned.

If *filno* is a key number whose associated data file has variable length records, then only the minimum record length portion (defined by *datlen* in the original call to **CreateDataFile**) is actually read into the buffer pointed to by *recptr*. If you wish to read the entire variable length record into the same or a different buffer, issue a call to **ReReadVRecord** after the call to **PreviousRecord**. Note that **ReReadVRecord** requires you to specify the size of the buffer area so that it can check if sufficient space is available.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful update of current ISAM record.
33	DNUL_ERR	*recptr* is NULL. No data file read performed.
42	DLOK_ERR	Could not get lock on data record. No data file read performed.

Value	Symbolic Constant	Explanation
48	FMOD_ERR	**PreviousRecord** called with a data file number of a variable length data file.
100	ICUR_ERR	No current ISAM record.
101	INOT_ERR	**PreviousRecord** not found. Either current ISAM record is the first record or its key value for index file *filno* is missing. Current ISAM record is not updated.

EXAMPLE

```
COUNT  acct_dat,acct_idx;
TEXT recbuf[320];

   if (LastRecord(acct_idx,recbuf) == NO_ERROR)
     ReReadVRecord(acct_dat,recbuf,320);
   while (isam_err == NO_ERROR) {
     post_interest(recbuf);
     if (PreviousRecord(acct_idx,recbuf) == NO_ERROR)
         ReReadVRecord(acct_dat,recbuf,320);
   }
```

LIMITATIONS

No check is made to determine if *recptr* points to a region sufficiently large to accept a data record. If the area is too small, either code or data will be clobbered.

SEE ALSO

NextRecord, **ReReadVRecord** and **PreviousInSet**

PutDODA

Assign the contents of a DODA to a data file.

SHORT NAME PUTDODA

TYPE Resource function.

DECLARATION

```
COUNT PUTDODA(datno,doda,numfld)
COUNT     datno;
DATOBJ   *doda;
UCOUNT    numfld;
```

DESCRIPTION **PutDODA** assigns the contents of a data object definition array (DODA) to a data file. The DODA is converted into a schema map and schema names as described in Appendix PRG D. Resources must be enabled for the data file. *doda* points to the beginning of the DODA (i.e., the array of DATOBJ's). The DATOBJ is discussed in Appendix PRG D. The *numfld* parameter indicates how many data fields are in the DODA.

RETURN

Value	Symbolic Constant	Explanation
48	FMOD_ERR	only valid for data files.
62	LERR_ERR	the file must be opened exclusively.
401	RNON_ERR	resources not enabled.
448	DEFP_ERR	file definition permission denied.

SEE ALSO **GetDODA.**

ReadData

Read data record.

SHORT NAME REDREC

TYPE Low level data file function

DECLARATION

```
COUNT ReadData(datno,recbyt,recptr)
COUNT   datno;
LONG recbyt;
VOID   *recptr;
```

DESCRIPTION **ReadData** reads the record at position *recbyt* of data file *datno* into the buffer area pointed to by *recptr*. If *datno* refers to a variable record length file, then **ReadData** reads only the minimum record length portion (defined by *datlen* in **CreateDataFile**) into the buffer. Use **ReReadVRecord** to read an entire variable length record.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful read.
29	ZREC_ERR	recbyt is zero.
30	LEOF_ERR	*recbyt* exceeds the logical end-of-file maintained in the data file header.
33	DNUL_ERR	recptr is null.
35	SEEK_ERR	lseek failed while preparing for read.
36	READ_ERR	Operating system could not execute read.

EXAMPLE

```
COUNT    datno,keyno;
TEXT     recptr[1024];
LONG     part_number;

scanf("%ld",&part_number);
if (ReadData(datno,GetKey(keyno,&part_number),recptr))
   printf("\nCould not retrieve record for part #%ld",
     part_number);
```

SEE ALSO **WriteData** and **ReReadVRecord**

ReadVData

Read variable length data record.

SHORT NAME RDVREC

TYPE Low level variable record length function

DECLARATION

```
COUNT ReadVData(datno,recbyt,recptr,varlen)
COUNT   datno;
LONG    recbyt;
VOID   *recptr;
VRLEN  varlen;
```

DESCRIPTION **ReadVData** reads the variable length data record at byte position *recbyt* for data file *datno* into the buffer area pointed to by *recptr*. If *varlen*, the size of the buffer pointed to by *recptr*, is not large enough to accommodate the variable length record, **ReadVData** returns an error.

If you wish to ensure that *varlen* is large enough, call **VDataLength** before your call to **ReadVData**. **VDataLength** returns the actual length of the record. If the existing buffer is not large enough, you can allocate a new, larger buffer before calling **ReadVData**.

NOTE: No check is actually made to be sure that the region pointed to by *recptr* is in fact as large as *varlen* indicates. It is up to your program to ensure that *varlen* accurately specifies the size of the area pointer to by *recptr*.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful read.
29	ZREC_ERR	Attempt to read at byte offset zero.
33	DNUL_ERR	*recptr* is null.
35	SEEK_ERR	lseek failed while preparing for read.
36	READ_ERR	Operating system could not execute read.
48	FMOD_ERR	*datno* is not assigned to a variable length data file.
123	RVHD_ERR	Record is not preceded by a valid record mark.
153	VBSZ_ERR	*varlen* < actual record length.
154	VRCL_ERR	variable length record is zero bytes long.

Value	Symbolic Constant	Explanation
158	VFLG_ERR	variable length record is not marked as active.

EXAMPLE

```
COUNT   datno,keyno;
VRLEN   cur_bufsiz,varlen;
TEXT    *recptr,*calloc();
LONG    part_number,pntr;

scanf("%ld",&part_number);
if ((pntr = GetKey(keyno,&part_number)) != DRNZERO) {
   varlen = VDataLength(datno,pntr);
   if (varlen > cur_bufsiz) {
     free(recptr);
     recptr     = calloc(1,varlen));
     cur_bufsiz = varlen;
   }
   if (ReadVData(datno,pntr,recptr,varlen))
     printf("\nError #%d reading at %ld",uerr_cod,pntr);
}
```

LIMITATIONS

If you dynamically allocate buffer space, remember to free the space when the buffer is no longer required.

SEE ALSO

ReReadVRecord, **VDataLength** and **WriteVData**

RebuildIFile

Incremental ISAM rebuild.

SHORT NAME RBLIFIL

TYPE ISAM function

DECLARATION

```
COUNT RebuildIFile(ifilptr)
IFIL *ifilptr;
```

DESCRIPTION **RebuildIFile** rebuilds the data file and (optional) index file referenced by the IFIL structure pointed to by *ifilptr*. When using an incremental ISAM file, all indices associated with the data file must be contained in a single, companion index. The file name pointed to by the IFIL structure (see ifilptr->pfilnam in Chapter PRG 5) SHOULD NOT include an extent: ".dat" and ".idx" are automatically appended to the file name for the data file and index file, respectively.

If you desire the files to be rebuilt in directories that are not specified until run-time, then the file name pointed to by the IFIL structure should not be finalized until run-time.

Consecutive file numbers are assigned to the data file and any (optional) indices associated with the data file. The IFIL structure contains two file numbers: a permanent file number (dfilno), and a temporary file number (tfilno). If dfilno is less than zero, then c-tree Plus finds the first available block of consecutive file numbers which can accommodate the data file and its indices. If such a block of numbers is found, then the first number in the block is assigned to the data file and to tfilno. If dfilno is greater than or equal to zero, then dfilno is assigned to the data file and to tfilno.

See Chapter PRG 5 for a complete description of the IFIL structure.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful rebuild of ISAM files.
12	FNOP_ERR	Could not open file(s).
20	KMIN_ERR	Key length too large for node size.
22	FNUM_ERR	File number is out of range.
45	KLEN_ERR	Key length exceeds MAXLEN parameter in *ctoptn.h*.

Value	Symbolic Constant	Explanation
46	FUSE_ERR	If *ifilptr->dfilno* >= 0, then file number (range) already in use. If *ifilptr->dfilno* < 0, then no satisfactory block of numbers available.
107	IDRK_ERR	Too many index files for one data file. Increase MAX_DAT_KEY parameter in *ctoptn.h*.
109	IKRS_ERR	Too many key segments. Increase MAX_KEY_SEG parameter in *ctoptn.h*.
115	ISLN_ERR	Key segments do not equal key length.
120	RRLN_ERR	Not enough dynamic memory for record buffer
121	RSPC_ERR	Not enough dynamic memory for sort buffer area. Change SORT_SPACE in *ctibld.c*.
122	RMOD_ERR	Attempt to change between fixed and variable length records.
123	RVHD_ERR	A variable length record is not preceded by a valid record mark.

EXAMPLE

```
extern IFIL  customer;
extern COUNT data_no;

if (OpenIFile(&customer)) {
   printf("\nCould not open %s with Error #%d.",
     customer.pfilnam,isam_err);
   if (isam_err == FCRP_ERR || isam_err == FCMP_ERR) {
       printf(" Attempting a rebuild.\n");
       if (RebuildIFile(&customer)) {
     printf("Rebuild failed with Error #%d\n",
         isam_err);
     exit(2);
       } else {
      printf("\nRebuild succeeded.\n");
      if (OpenIFile(&customer)) {
         printf("\nUNEXPECTED ERROR ON REOPEN #%d\n",
             isam_err);
         exit(2);
      }
       }
   }
}
datno = customer.tfilno;
```

LIMITATIONS

RebuildIFile cannot be used during multi-user (or LAN) operations. The rebuild facility is designed for off-line processing.

SEE ALSO

RebuildFileXtd, **InitISAM** and Chapter PRG 5 which describes the ISAM parameters.

RebuildIFileXtd

Incremental ISAM rebuild, extended.

SHORT NAME RBLIFILX

TYPE Extended ISAM function

DECLARATION

```
COUNT
RebuildIFileXtd(ifilptr,dataextn,indxextn,permmask,
        groupid,fileword)
IFIL *ifilptr;
TEXT  *dataextn;
TEXT  *indxextn;
LONG   permmask;
TEXT  *groupid;
TEXT  *fileword;
```

DESCRIPTION

RebuildIFileXtd is a variation of **RebuildIFile** that permits the use of the FairCom Server's security system. In addition, you can use it to change the file name extents from the defaults that are used in **RebuildIFile**. Please read the description of **RebuildIFile** before proceeding with this description, as we will only describe the extended features here.

permmask is the **permission mask** that will be assigned to the files created here. It is formed by "OR"ing the appropriate permission constants. For more information on permission masks refer to page PRG 12-3.

groupid is a pointer to a buffer that contains the **group id** that this file is to be assigned to. The group id must be valid for the user that is creating the file. If *groupid* is null, the file will be assigned to the default group for the user. See Chapter PRG 12 for more information on groups.

fileword is an optional **file password**, as explained on page PRG 12-4. If *fileword* is null then there will be no password for this file. If one is established then every user will need to use the password to be able to open the file.

dataextn and *indxextn* are pointers to buffers that can be used to specify optional data and index file name extensions, respectively. The extensions are 8 byte ASCIIZ strings. If they are null pointers then the default extension will be used, ".dat" for data files and ".idx" for index files. If you want no extension then you should pass a pointer to a buffer that contains only blanks terminated by a null character. Do not set both extensions to blanks.

RETURN The following error codes may be seen in addition to those shown for **RebuildIFile**.

Value	Symbolic Constant	Explanation
455	SGRP_ERR	This user does not belong to the group *groupid*.

ReleaseData

Return fixed length data record.

SHORT NAME RETREC

TYPE Low level data file function

DECLARATION

```
COUNT ReleaseData(datno,recbyt);
COUNT   datno;
LONG recbyt;
```

DESCRIPTION **ReleaseData** adds record position *recbyt* to the chain of deleted records for fixed record length file *datno*. **ReleaseData** should be called when a data record is no longer needed so that it can be reused. Records returned by **ReleaseData** are reused by **NewData** before the data file is extended. **ReleaseData** is automatically called by **DeleteRecord**.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful return operation.
29	ZREC_ERR	*recbyt* is zero.
30	LEOF_ERR	*recbyt* exceeds the logical end-of-file maintained in the data file header.
32	DDRN_ERR	Attempt to return *recbyt* twice in a row.
48	FMOD_ERR	*datno* is not assigned to a fixed length data file.
57	DADV_ERR	Proper lock not found by the FairCom Server. See *ctsrvr.c*

EXAMPLE

```
COUNT         keyno,datno;
POINTER       recbyt;
TEXT         *target;

if (recbyt = GetKey(keyno,target) &&
   DeleteKey(keyno,target,recbyt) == NO_ERROR) &&
   ReleaseData(datno,recbyt) == NO_ERROR)
     printf("\nSUCCESS.");
```

LIMITATIONS **ReleaseData** writes an 0xff byte and a four byte data record position starting at the first byte of the returned data record. Therefore, even if you only write ASCII

information into the data records, **ReleaseData** will place binary information into the beginning of deleted records.

SEE ALSO **NewData**

ReleaseVData

Return variable length data record.

SHORT NAME RETVREC

TYPE Low level data file function

DECLARATION

```
COUNT ReleaseVData(datno,recbyt);
COUNT   datno;
LONG    recbyt;
```

DESCRIPTION **ReleaseVData** adds record position *recbyt* to the pool of available records for variable record length file *datno*. **ReleaseVData** should be called when a data record is no longer needed so that it can be reused. Records returned by **ReleaseVData** are reused by **NewVData** before the data file is extended, provided that they are long enough to accommodate the **NewVData** request. **ReleaseVData** is automatically called by **DeleteVRecord**.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful return operation.
2	KDUP_ERR	Attempt to delete a record already deleted.
48	FMOD_ERR	*datno* is assigned to a fixed length data file.
159	VPNT_ERR	*recbyt* is zero.
57	DADV_ERR	Proper lock not found by the FairCom Server. See *ctsrvr.c*

EXAMPLE

```
COUNT     keyno,datno;
LONG      recbyt;
TEXT     *target;

if (recbyt = GetKey(keyno,target) &&
   DeleteKey(keyno,target,recbyt) == NO_ERROR) &&
   ReleaseVData(datno,recbyt) == NO_ERROR)
     printf("\nSUCCESS.");
```

SEE ALSO **NewVData**

ReReadRecord

(Re)read current ISAM record.

SHORT NAME RRDREC

TYPE ISAM function

DECLARATION

```
COUNT ReReadRecord(datno,recptr)
COUNT datno;
VOID *recptr;
```

DESCRIPTION **ReReadRecord** provides a simple way to (re)read the current ISAM record for fixed length data records belonging to data file number *datno*. The records are read into the record buffer pointed to by *recptr*. **ReReadVRecord** should be used to (re)read the current ISAM record for a variable length data file unless it is desired only to reread the fixed length portion of the variable length record.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful read current ISAM record.
33	DNUL_ERR	*recptr* is NULL. No data file read performed.
42	DLOK_ERR	Could not get lock on data record. No data file read performed.
100	ICUR_ERR	No current ISAM record.
160	ITIM_ERR	Record deleted by another user.

EXAMPLE

```
COUNT datano;
TEXT buffer[256];

if (LockISAM(ENABLE) || ReReadRecord(datano,buffer))
{  printf("\nCould not reread record (%d)",isam_err);
   LKISAM(FREE);
   return(-1);
}
```

SEE ALSO **ReReadVRecord**, the LOCK discussion of multi-user updates in Chapter PRG 6, and the source code of CTEXMG.C.

ReReadVRecord

(Re)Read variable length data record.

SHORT NAME REDVREC

TYPE ISAM variable record length function

DECLARATION

```
COUNT ReReadVRecord(datno,recptr,varlen)
COUNT   datno;
VOID   *recptr;
VRLEN  varlen;
```

DESCRIPTION

ReReadVRecord reads the current variable length ISAM record for data file *datno* into the buffer area pointed to by *recptr*. If *varlen*, the size of the buffer pointed to by *recptr*, is not large enough to accommodate the variable length record, **ReReadVRecord** returns an error. **ReReadVRecord** is used to read the entire variable length record after the fixed length has been read via one of the ISAM search routines (e.g., **FirstInSet** or **NextRecord**). **ReReadVRecord** also can be used to reread the current ISAM variable length record.

If you wish to ensure that *varlen* is large enough, call **VRecordLength** before your call to **ReReadVRecord**. **VRecordLength** returns the actual length of the record. If the existing buffer is not large enough, you can allocate a new, larger buffer before calling **ReReadVRecord**.

NOTE: No check is actually made to be sure that the region pointed to by *recptr* is in fact as large as *varlen* indicates. It is up to your program to ensure that *varlen* accurately specifies the size of the area pointer to by *recptr*.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful read.
33	DNUL_ERR	*recptr* is null.
35	SEEK_ERR	lseek failed while preparing for read.
36	READ_ERR	Operating system could not execute read.
48	FMOD_ERR	*datno* is not assigned to a variable length data file.
100	ICUR_ERR	No current ISAM record for file *datno*.
153	VBSZ_ERR	*varlen* < actual record length.
154	VRCL_ERR	variable length record is zero bytes long.

Value	Symbolic Constant	Explanation
158	VFLG_ERR	variable length record is not marked as active.
159	VPNT_ERR	*recbyt* is zero.

EXAMPLE

```
COUNT   datno,keyno;
VRLEN   cur_bufsiz,varlen;
TEXT    *recptr,minlen_buffer[192],*calloc();
LONG    part_number;

scanf("%ld",&part_number);
if (GetRecord(keyno,&part_number,minlen_buffer) ==
          NO_ERROR) {
   varlen = VRecordLength(datno);
   if (varlen > cur_bufsiz) {
     free(recptr);
     recptr     = calloc(1,varlen));
     cur_bufsiz = varlen;
   }
   ReReadVRecord(datno,recptr,varlen);
}
```

LIMITATIONS

If you dynamically allocate buffer space, remember to free the space when the buffer is no longer required.

SEE ALSO

ReadVData, **ReReadRecord**, **VRecordLength** and **WriteVData**

ResetRecord

This function updates the current ISAM record image buffers.

SHORT NAME UPDCURI

TYPE ISAM function

DECLARATION

```
COUNT ResetRecord(datno,mode);
COUNT datno;
COUNT mode;
```

DESCRIPTION

ResetRecord manipulates the internal image of the current ISAM record kept c-tree Plus. An image is kept of the current ISAM record's key structure, as well as the current ISAM record's file address. Whenever you perform an ISAM function that changes the current ISAM record, such as an **AddRecord** or **ReWriteRecord**, the original current ISAM record information is copied over to a secondary buffer before the primary buffer is changed. **ResetRecord** can be used to manage these two buffers.

datno specifies the ISAM file that is to be affected. *mode* can be one of several values;

- SWTCURI - This will switch the ISAM buffers. Use this value of *mode* after a rewrite call to move the old buffer back to the original. This allows you to keep your position in the file as it was before the rewrite.
- SAVCURI - This copies the current ISAM record information to the secondary buffer.
- RSTCURI - This restores the primary ISAM record information from the secondary area.

RETURN

Value	Symbolic Constant	Explanation
22	FNUM_ERR	File number is out of range.
26	FACS_ERR	File number is not assigned to a file in use.
47	FINT_ERR	c-tree has not been initialized.
48	FMOD_ERR	*datno* is not assigned to a data file.
116	IMOD_ERR	invalid value in *mode*.

EXAMPLE

```
COUNT          invfil;
struct invd {
   TEXT        delflg;
   LONG        part_no;
   TEXT        part_name[60];
   LONG        on_hand;
} recbuf;

scanf("%ld %59s %ld",&recbuf.part_no,recbuf.part_name,
   &recbuf.on_hand);
recbuf.delflg = '\0'; /* clear delete flag */
if (AddRecord(invfil,&recbuf) )
   printf("\nAddRecord error %d in file %d",
          isam_err,isam_fil);
/* after adding record, restore current ISAM to
 * previous point */
if (ResetRecord(invfil,SWTCURI) )
   printf("\nCannot reset, error %d",isam_err);
```

SEE ALSO **SetRecord**

RestoreSavePoint

Undo transaction operations back to a savepoint.

SHORT NAME TRANRST

TYPE Low level data file function

DECLARATION

```
COUNT RestoreSavePoint(savpnt);
COUNT savpnt;
```

DESCRIPTION

This function is used to back the system up to a previously defined savepoint, as created with a call to **SetSavePoint**. The parameter *savpnt* is the savepoint to back up to, and is the value returned by the call to **SetSavePoint**. This allows you to back up in a transaction to a particular point, without having to abort or commit the entire transaction. For a complete discussion of this process please see Chapter PRG 11.

savpnt can also be specified as a small negative number. -1 means go back to the most current savepoint. -2 means go back one more, etc.

RestoreSavePoint also clears errors which have occurred since the savepoint.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	no error occurred.
71	TNON_ERR	there is no active transaction pending

EXAMPLE

```
COUNT savepoint;

VOID domaster()
{Begin(ENABLE|TRNLOG);                  /*start transaction with locks*/
 while(another());                      /* get next record to add */
 { savepoint = SetSavePoint();          /* get save point at beginning
                                           of each master record */
   if (add_master()<0) Abort(); /*abort if can't add master rec.*/
   dodetail();                          /* process detail records */
 }
 if (Commit(FREE)>0) printf("\nError %d in transaction",uerr_cod);
 return;
}
```

```
VOID dodetail()
{while(moredetail());              /* get next detail record to add */
 { if (add_detail()<0)             /* add details, if possible */
   { RestoreSavePoint(savepoint) /* with error, return to savept*/
     return;
   }
 }
}
```

SEE ALSO **SetSavePoint**

ReWriteRecord

Rewrite current fixed length record.

SHORT NAME RWTREC

TYPE ISAM function

DECLARATION

```
COUNT ReWriteRecord(datno,recptr)
COUNT datno;
VOID *recptr;
```

DESCRIPTION

ReWriteRecord allows changes to be made in the current fixed length ISAM record for data file *datno*. *recptr* should point to the updated version of the current ISAM record. **ReWriteRecord** assumes that the current ISAM record is the original version. After successful completion of **ReWriteRecord**, the updated version is now stored in the data file and all necessary changes to the key values associated with the updates are automatically made.

The typical **ReWriteRecord** usage is: first, an ISAM retrieval function (e.g., **GetRecord** or **NextRecord**) reads the desired record, making it the current ISAM record. After updating the record buffer, call **ReWriteRecord**. The updated record becomes the current ISAM record. If any of the key fields have been changed, the old key is removed from the appropriate indices, and the new one is added.

Unlike c-tree version 4.3 and older, you do not have to be careful about making changes to the current record in your buffer. c-tree Plus maintains its own copy of the current ISAM record key information and position.

Multi-user updates are more complex because of the possibility of either locking out the record for prolonged periods, or because of two users trying to update the same record at the same time. See Chapter PRG 6 for a detailed discussion of multi-user updates.

ReWriteRecord does not change the automatic serial number key segments, if any. If you want the serial number updated, delete the old record and add the updated record.

To make the old record the current ISAM record (so that a **NextRecord** will be performed relative to the position before the update), make the following call after successfully completing the record rewrite:

```
ResetRecord(datno,SWTCURI);
```

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful record update. New record becomes the current ISAM record.
2	KDUP_ERR	Duplicate key value detected in index file number isam_fil. Record update not performed.
22	FNUM_ERR	File number is out of range.
26	FACS_ERR	File number is not assigned to a file in use.
48	FMOD_ERR	*datno* is not assigned to a fixed length data file.
57	DADV_ERR	Proper lock not found by the c-tree Plus . See *ctsrvr.c*
100	ICUR_ERR	No current ISAM record.
105	IUND_ERR	Could not undo a rejected ISAM update. This is a serious error indicating that the data file must be rebuilt. This does not apply to files supporting transaction processing.
199	NSCH_ERR	key segment refers to schema, but schema not defined.
433	SSCH_ERR	segment definition inconsistent with schema.
445	SDAT_ERR	not enough data to assembly key.
446	BMOD_ERR	invalid key mode.

EXAMPLE

```
COUNT datno,keyno;
LONG  cust_no;
TEXT  buffer[128];

printf("\nEnter customer number: ");
scanf("%ld",&cust_no);
if (GetRecord(keyno,buffer))  /* read current version
                    into buffer */
   printf("\nCould not read customer record.");
else {
   update(buffer); /* make your changes to record */
   if (ReWriteRecord(datno,buffer))
     printf("\nCould not rewrite customer record.");
   else
     printf("\nSuccessful update of customer record.");
}
```

LIMITATIONS

After a call to **ReWriteRecord**, the current ISAM record is set to the new record. This may create a situation that you don't expect when **ReWriteRecord** is used inside a loop which steps over the data in key sequential order. In particular, if the key value is changed during the update, the current ISAM record position will skip to the new position which may be before or after the original position. To avoid this situation, you can reset the current ISAM record after the call to **ReWriteRecord** by a call to **ResetRecord** as mentioned above.

ReWriteVRecord

Rewrite current variable length record.

SHORT NAME RWTVREC

TYPE ISAM function

DECLARATION

```
COUNT ReWriteVRecord(datno,recptr,varlen)
COUNT  datno;
VOID  *recptr;
VRLEN varlen;
```

DESCRIPTION

ReWriteVRecord allows changes to be made in the current variable length ISAM record for data file *datno*. *recptr* should point to the updated version of the current ISAM record, and *varlen* specifies the length of the updated version. **ReWriteVRecord** assumes that the current ISAM record is the original version. After successful completion of **ReWriteVRecord**, the updated version is now stored in the data file and all necessary changes to the key values associated with the updates are automatically made.

If the updated version does not fit into the space available for the current record, it is moved to another location in the data file. All associated indices are updated appropriately.

The typical **ReWriteVRecord** usage is: first, an ISAM retrieval function (e.g., **GetRecord** or **NextRecord**) brings a portion of the desired record into a buffer area. Then **ReReadVRecord** is used to read the entire record into the buffer. This buffer now contains the current ISAM record. When **ReWriteVRecord** is called, the *recptr* parameter should point to the buffer area which holds the updated version.

NOTE: Multi-user updates are more complex because of the possibility of either locking out the record for prolonged periods, or because of two users trying to update the same record at the same time. See Chapter PRG 6 for a detailed discussion of multi-user updates.

NOTE: **ReWriteVRecord** does not change the automatic serial number key segments, if any. If you want the serial number updated, delete the old record and add the updated record.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful record update. New record becomes the current ISAM record.
2	KDUP_ERR	Duplicate key value detected in index file number isam_fil. Record update not performed.
22	FNUM_ERR	File number is out of range.
26	FACS_ERR	File number is not assigned to a file in use.
48	FMOD_ERR	*datno* is not assigned to a variable length data file.
57	DADV_ERR	Proper lock not found by FairCom Server. See *ctsrvr.c*
100	ICUR_ERR	No current ISAM record.
105	IUND_ERR	Could not undo a rejected ISAM update. This is a serious error indicating that the data file must be rebuilt.
199	NSCH_ERR	key segment refers to schema, but schema not defined.
433	SSCH_ERR	segment definition inconsistent with schema.
445	SDAT_ERR	not enough data to assembly key.
446	BMOD_ERR	invalid key mode.

EXAMPLE

```
COUNT  datno,keyno;
VRLEN newlen,varlen;
LONG   cust_no;
TEXT   buffer[512];

printf("\nEnter customer number: ");
scanf("%ld",&cust_no);
    /* read current version into buffer */
if (GetRecord(datno,buffer) ||
       ReReadVRecord(datno,buffer,512))
   printf("\nCould not read customer record.");
else {
   newlen = update(buffer); /* create update in buffer
   if (ReWriteVRecord(datno,buffer,newlen))
     printf("\nCould not rewrite customer record.");
   else
     printf("\nSuccessful update of customer record.");
}
```

LIMITATIONS

After a call to **ReWriteVRecord**, the current ISAM record is set to the new record. This may cause counter-intuitive results when **ReWriteVRecord** is used inside a loop which steps over the data in key sequential order. In particular, if the key value is changed during the update, the current ISAM record position will skip to the new position which may be before or after the original position. To avoid this situation you can reset the current ISAM record after the call to **ReWriteVRecord** is completed. Call **ResetRecord**(datno,SWTCURI) to restore the current ISAM record information to the original position.

SEE ALSO

Chapter PRG 6 for details on multi-user updates.

Security

Used to modify a file's password, group ID, and permission mask.

SHORT NAME SECURITY

TYPE Low Level

DECLARATION

```
COUNT SECURITY(filno,bufptr,bufsiz,mode)
COUNT filno;
VOID *bufptr;
VRLEN bufsiz;
COUNT mode;
```

DESCRIPTION

Security permits a file's password, Group ID, and permission mask to be modified. **Security** will only permit the the owner of a file to change the file's security information. Otherwise BOWN_ERR is returned.

bufptr points the buffer which contains the information to update the security information, and *bufsiz* specifies the length of the buffer. The buffer should contain the old password (null terminated) immediately followed by the new information (also null terminated), which can be the new file password, new file group ID, or the new permission mask. If you are changing the permission mask then the value following the null terminated old password must be a long integer containing the appropriate ORed permission values.

To remove a password, simply pass a null byte for the new password.

mode defines the type of request. The values for *mode* are:

Symbolic Constant	Value	Interpretation
SEC_USERWORD	0	change user password.
SEC_FILEWORD	1	change file password.
SEC_FILEGRUP	2	change file group.
SEC_FILEMASK	3	change file permission mask.

RETURN

Value	Symbolic Constant	Explanation
22	FNUM_ERR	File number is out of range.
47	FINT_ERR	c-tree has not been initialized.

Value	Symbolic Constant	Explanation
48	FMOD_ERR	invalid *mode* value
62	LERR_ERR	file must be opened exclusively
447	BOWN_ERR	only the user/owner can change the user/file password.
448	DEFP_ERR	file definition permission denied
451	LPWD_ERR	the (old) password is not valid.
455	SGRP_ERR	the file owner does not belong to the new group

SEE ALSO See Chapter PRG 12 for more information.

SetAlternateSequence

Allows you to establish an Alternate Collating Sequence for an index file.

SHORT NAME SETALTSEQ

TYPE Resource function

DECLARATION

```
COUNT SetAlternateSequence(keyno,altseq);
COUNT keyno, *altseq;
```

DESCRIPTION **SetAlternateSequence** assigns an alternative collating sequence to index *keyno* which has just been created (and not yet closed) or has been opened exclusively. Resources must be enabled for the index file in order to successfully complete this function.

altseq points to an array of 256 short integers which represent the new collating sequence. The zeroeth and 255th entries in the array must be zero and 255, respectively. All other entries must be between zero and 255, inclusively. The value in the array determines where the underlying byte value will be mapped: a high value inplies toward the end of the sequence.

For example, to switch the position of "A" and "B" in the collating sequence, then

```
altseq['A'] = 'B';
altseq['B'] = 'A';
```

Although this is a low level function, it ordinarily only applies to ISAM level key operations in which the segment mode can specify the use of an alternative collating sequence.

RETURN

Value	Symbolic Constant	Explanation
22	FNUM_ERR	File number is out of range.
47	FINT_ERR	c-tree has not been initialized.
48	FMOD_ERR	does not apply to data files
62	LERR_ERR	file must be opened exclusively
401	RNON_ERR	resources not enabled.
448	DEFP_ERR	file definition permission denied

SEE ALSO **GetAlternateSequence.**

SetRecord

Allows resetting of the current ISAM record.

SHORT NAME SETCURI

TYPE ISAM function

DECLARATION

```
COUNT SetRecord(datno,recbyt,recptr,datlen);
COUNT datno;
LONG recbyt;
VOID *recptr;
VRLEN datlen;
```

DESCRIPTION

c-tree Plus maintains the record number of the current ISAM record for each data file, as well as an image of the key buffer. There may be times when you want to reset the current ISAM record. For instance, if you want to return to a particular point in the file and index in order to continue a sequential scan of the file. By calling **SetRecord**, you can set the current record number of file *datno* to be the value in *recbyt*. *recptr* points to a record image. If *datlen* is zreo, then *recptr* should point to a complete fixed length record, or the fixed portion of a variable length record. Otherwise, *datlen* indicates the size of the information pointed to by *recptr*.

RETURN

Value	Symbolic Constant	Explanation
22	FNUM_ERR	File number is out of range.
26	FACS_ERR	File number is not assigned to a file in use.
47	FINT_ERR	c-tree has not been initialized.
48	FMOD_ERR	*datno* is not assigned to a data file.

EXAMPLE

```
COUNT datno,keyno;
LONG  custno,recpos;
TEXT  savbuf[128],buffer[128];

printf("\nEnter customer number: ");
scanf("%ld",&custno);
if (GetRecord(keyno,savbuf))
   printf("\nCould not read customer record");
else{
   recpos = CurrentFileOffset(datno);
   process_more_records_in(buffer);
```

```
/* restore original record as current ISAM */
SetRecord(datno,recpos,savbuf,128L);
```

SEE ALSO **CurrentISAMKey, TransformKey**

SetSavePoint

Establish a save point.

SHORT NAME TRANSAV

TYPE Low level data file function

DECLARATION

```
COUNT SetSavePoint();
```

DESCRIPTION This function is used to create a save point in a transaction. Often when you are processing a complex transaction you will find times when you want to back up just part way in that transaction. You will not want to use **Commit**, as that will commit the entire transaction, as well as ending it. You will not want to use **Abort**, as that aborts the entire transaction.

To be able to reverse just a portion of a transaction, you call **SetSavePoint** at the point that you may want to return to. This is a save point. There can be any number of save points within a transaction. The **SetSavePoint** call returns a COUNT value that is the numerical representation of that save point. At a later point, if you want to back up the transaction to a particular save point, you will issue a call to **RestoreSavePoint**, passing the specific save point value as the parameter.

RETURN The return value of **SetSavePoint** is the numerical representation of the save point. Use this value in a subsequent call to **RestoreSavePoint** to reset the transaction to this save point. If an error occurs, **SetSavePoint** returns zero, and you must check *uerr_cod* for the error value.

Value	Symbolic Constant	Explanation
71	TNON_ERR	there is no active transaction pending
94	PNDG_ERR	An update error has occured during one of the previous operations which has not been cleared through **RestoreSavePoint** or **ClearTranError**.

EXAMPLE

```
COUNT savepoint;

VOID domaster()
{Begin(ENABLE|TRNLOG);                  /*start transaction with locks*/
 while(another());                      /* get next record to add */
 { savepoint = SetSavePoint();          /* get save point at beginning
                                           of each master record */
   if (add_master()<0) Abort(); /*abort if can't add master rec.*/
```

```
        dodetail();                    /* process detail records */
      }
      if (Commit(FREE)>0) printf("\nError %d in transaction",uerr_cod);
      return;
    }

    VOID dodetail()
    {while(moredetail());             /* get next detail record to add */
     { if (add_detail()<0)            /* add details, if possible */
       { RestoreSavePoint(savepoint) /* with error, return to savept*/
         return;
       }
     }
    }
```

SEE ALSO **RestoreSavePoint**

SetVariableBytes

Change the values to be used for key padding and as a field delimiter in key construction.

SHORT NAME SETVARBYTS

TYPE Resource function

DECLARATION

```
COUNT SetVariableBytes(filno, resptr);
COUNT filno;
VOID  *resptr;
```

DESCRIPTION

When called for a data file, this function changes the value used to pad varying length key segments as well as the byte value used to delimit varying length string fields (which are not length count delimited).

When called for an index file, this function changes the padding byte used during trailing padding key compression.

The default values (if this function is not used) are blanks for padding, and null for the field delimiter.

resptr is a pointer to a 2 byte array that contains the values to be used for padding (first byte) and as a field delimiter (second byte). This information will be stored in the file as a resource, therefore resources must be enabled for the file.

filno must be opened with an EXCLUSIVE file mode for it to be successful.

For a related set of data and index files, **SetVariableBytes** should be called for the data and index files immediately after successful creation.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	no error occurred.
63	LERR_ERR	the file must be opened with an EXCLUSIVE lock.
401	RNON_ERR	Resources are not enabled.

EXAMPLE

```
COUNT filnum,thebytes[2];

if (OpenDataFile(filnum,"test.dat",
        EXCLUSIVE | PERMANENT)
```

```
      printf("\nCannot open file (%d)",uerr_cod);
else
{  thebytes[0] = 00;  /*set padding byte */
   thebytes[1] = 32;  /*set delimiter byte */
   if (setVariableBytes(filnum,&thebytes)
       printf("\nByte error %d",uerr_cod);
}
```

StopUser

Log application off of the FairCom Server.

SHORT NAME STPUSR

TYPE Low level function

DECLARATION

```
COUNT StopUser()
```

DESCRIPTION

StopUser logs an application off a FairCom Server. It causes all the files not yet closed by the application to be closed, and frees a slot in the server for another user. **StopUser** is automatically invoked by **CloseISAM**(). Therefore, **StopUser** is only required in low level applications, or in ISAM applications when a program can exit without calling **CloseISAM**.

NOTE: **StopUser** is ignored in non-server applications.

NOTE: a c-tree Plus application is automatically logged on to a server whenever **InitCTree**, **InitISAM**, **OpenISAM** or **CreateISAM** is called. There is no separate call for logging-on to the server.

RETURN

StopUser returns an error if it could not communicate with the server. This might happen if the server has been shut down or if the communications link has been disturbed.

Value	Symbolic Constant	Explanation
133	ASKY_ERR	Server cannot be found (has it been shut down?).
162	SGON_ERR	Server communication link has been disturbed.
410	USTP_ERR	user is not logged on when **StopUser** was called.

EXAMPLE

```
TEXT *pfname;

if (OpenISAM(pfname)) {
   printf("\nCould not OpenISAM. Error %d. File %d.",
     isam_err,isam_fil);
   StopUser();
   exit(2);
}
```

TransformKey

Perform key segment translation to target key.

SHORT NAME TFRMKEY

TYPE ISAM level function

DECLARATION

```
TEXT *TransformKey(keyno,target)
COUNT keyno;
VOID *target;
```

DESCRIPTION

The ISAM parameters for your application can specify that key segments are to be translated (e.g., converted to upper case) before being concatenated and added to your index files. However, c-tree Plus does not automatically translate your target keys in the same way. **TransformKey** will convert a target key (pointed to by target) according to your ISAM parameters for index number *keyno*. For example, if you have reversed the bytes of a key segment (see segment mode equal to one in Chapter PRG 5), then passing a target key to **TransformKey** will cause the appropriate segment of the target key to be reversed.

Since **TransformKey** constructs the translated key in place, you must ensure that the key area (pointed to by target) is at least as large the key length (including suffix) defined for index number *keyno*. **TransformKey** expects that you have concatenated all the necessary segments into a target key pointed to by the target parameter.

NOTE: If you are using *single entry point function calls* (see Appendix PRG E), or the *extended functions* **CreateISAMXtd**, **OpenISAMXtd**, or **InitISAMXtd**, c-tree Plus will **automatically** invoke **TransformKey** for you (unless you disable this feature).

RETURN

TransformKey returns a pointer to the translated target key. Since the translation is performed in place, the return value is always equal to the second parameter (*target*) unless there is an error. If an index accepts duplicate key values, then **TransformKey** sets the suffix to zero in the target key. A NULL value is returned if there is an error, and the value of *isam_err* will be:

Value	Symbolic Constant	Explanation
80	BKEY_ERR	*target* is null or *keyno* is out of range.
199	NSCH_ERR	segment mode requires a schema map
433	SSCH_ERR	filed reference by schema based segment is not defined.

EXAMPLE

```
TEXT  alph_target[5],key_buffer[13],rec_buffer[750];
long  int_target;
short keyno;

/* target is comprised of a long integer, a five byte
  alphanumeric, and a duplicate key suffix. cpybuf
   copies n bytes (without regard to null bytes). */
cpybuf(key_buffer,&int_target,4);
cpybuf(key_buffer + 4,alph_target,5);
if (FirstInSet(keyno, TransformKey(keyno,key_buffer),
      rec_buffer,9))
   printf("\nUnsuccessful FirstInSet (%d)\n",isam_err);
```

LIMITATION

TransformKey automatically sets the suffix of the target key (if duplicates are enabled for index number *keyno*) to all null bytes. If you desire to control the suffix, you will have to change it after **TransformKey** is called.

If the target area pointed to by target is not as long as your key length, then **TransformKey** will clobber the memory area following the target key.

SEE ALSO

CurrentISAMKey

UpdateFileMode

Changes critical file mode attributes such as the level of transaction control.

SHORT NAME PUTFIL

TYPE Low Level function

DECLARATION

```
COUNT UpdateFileMode(filno,filmod);
COUNT filno;
COUNT filmod;
```

DESCRIPTION

When you create or open a file, the value of *filmod* sets the attributes for the file. Some of these attributes are permanent, assigned during the creation of the file. Others are temporary, assigned only while the file is open. **UpdateFileMode** allows you to change these attributes after the file has been opened or created.

filno specifies the file to be changed. This file must be currently open. *filmod* is the new file mode attribute. It must be perfectly formed, as it will **replace** the current file mode for this file.

Use this function with caution! It is possible to damage a file if you are not careful. For instance, changing a data file from transaction processing to no transaction processing could lead to a situation where the file becomes corrupted and automatic recovery is not available.

Some examples of changes that might make sense would be to add or delete CHECKLOCK, or to turn on one of the transaction modes for a data file, or to switch between PREIMG and TRNLOG for an index. You cannot go from a non-transaction index to a transaction index.

Note that *filno* must be opened in an EXCLUSIVE mode for the function to work.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	no error occured
48	FMOD_ERR	attempting to switch between fixed and variable length records.
63	LERR_ERR	file must be opened in EXCLUSIVE mode to modify.
99	TTYPE_ERR	cannot switch an index to transaction processing after it has been created.

EXAMPLE

```
COUNT filno;
if (UpdateFileMode(filno,TRNLOG))
   printf("\nFile Mode Error %d",uerr_cod);
```

LIMITATIONS

No check is made to determine if the change being made will damage the file or not.

UpdateResource

This function updates the value of the specified Resource record.

SHORT NAME UPDRES

TYPE Resource function

DECLARATION

```
COUNT UpdateResource(filno, resptr, varlen);
COUNT filno;
VOID  *resptr;
VRLEN varlen;
```

DESCRIPTION

UpdateResource is used to update the value of a particular Resource in file *filno*. *resptr* points to a Resource Data Block, as specified below, with *varlen* being the length of the Resource Data Block.

Byte Offset	Data Type	
0 - 3	unsigned long integer	Resource Type
4 - 7	unsigned long integer	Resource Number
8 - m	null terminated character string	Resource Name
m + 1 - n	any collection of data types desired	Resource Information

If a record lock has been placed on this resource via a prior call to **GetResource**, that lock is released by **UpdateResource**.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful update.
57	DADV_ERR	Proper lock not found by the FairCom Server.
401	RNON_ERR	Resources have not been enabled for this file.
408	RNOT_ERR	Resource not found. You cannot update a resource that doesn't exist.

EXAMPLE

```
#include "ctstdr.h"
#include "ctdecl.h"

structure {                          /* Resource Data Block structure */
   ULONG  resource_type;
   ULONG  resource_number;
   TEXT   variable_info[1016];    /* This will hold the Resource
                                     Name and Data*/
} my_resource;

if (GetResource(datno,"MY!resource",&my_resource,
                  (VRLEN)sizeof(my_resource),(RES_NAME|RES_LOCK)));
{   /*at this point you will update the information in the buffer*/
    if (UpdateResource(datno,&my_resource,sizeof(my_resource))
        printf("\nError %d when update resource.",uerr_cod);
    else
        printf("\nUpdate successful!");
} else
        printf("\nCould not get resource. Error #%d",uerr_cod);
```

SEE ALSO **GetResource, AddResource, DeleteResource**

VDataLength

Get length of variable length data record.

SHORT NAME

GTVLEN

TYPE

Low level variable length data file function

DECLARATION

```
VRLEN VDataLength(datno,recbyt)
COUNT   datno;
LONG recbyt;
```

DESCRIPTION

VDataLength returns the total length of the variable length data record beginning at byte offset *recbyt*.

RETURN

If the record length is zero or an error occurs, **VDataLength** will return a value of zero; otherwise, **VDataLength** returns the length of the variable length data record. If a zero value is returned, check the value of uerr_cod to determine if an error occurred. A non-zero value for uerr_cod indicates that an error has occurred.

Value	Symbolic Constant	Explanation
0	NO_ERROR	No error.
29	ZREC_ERR	**VDataLength** called with zero recbyt.
48	FMOD_ERR	*datno* is not assigned to a variable length data file.
158	VFLG_ERR	variable length record is not marked as active.

EXAMPLE

```
LONG pntr;
VRLEN  vreclen;
COUNT   datno,keyno;
TEXT    key_buffer[64];

if ((pntr = FirstKey(keyno,key_buffer)) != 0L) {
   vreclen = VDataLength(datno,pntr);
   if (uerr_cod == 0) {
     printf(
"\nFirst variable record by key requires %d bytes",
          vreclen);
   }
}
```

VRecordLength

Get length of current variable length ISAM record.

SHORT NAME GETVLEN

TYPE ISAM variable length record function

DECLARATION

```
VRLEN VRecordLength(datno)
COUNT datno;
```

DESCRIPTION **VRecordLength** returns the length of the current ISAM record for variable length data file *datno*. If an error occurs, **VRecordLength** returns a zero and isam_err is set to the error code. Ordinarily, **VRecordLength** is used after a call to an ISAM function (which reads only the minimum record length portion of the entire variable length record) in order to determine the buffer size requirements for the entire record. Then **ReReadVRecord** can be called. If you already know the maximum possible record length for the data file, it is not necessary to call **VRecordLength** before **ReReadVRecord**.

RETURN If no error occurs, **VRecordLength** returns the total record length for the current ISAM record. Otherwise, **VRecordLength** returns a zero and isam_err is set as follows:

Value	Symbolic Constant	Explanation
0	NO_ERROR	No error.
48	FMOD_ERR	*datno* is assigned to a fixed length data file.
100	ICUR_ERR	No current ISAM record for data file.

EXAMPLE

```
COUNT        invfil,datfil;
VRLEN       varlen;
struct invd  {
   LONG      part_no;
   TEXT      part_name[120];
} recbuf;

if (FirstRecord(invfil,&recbuf) == NO_ERROR &&
  (varlen = VRecordLength(datfil))
   printf("\nThe first record is %u bytes long.",varlen);
else
   printf("\nError #%d on file %d",isam_err,isam_fil);
```

SEE ALSO **ReReadVRecord** and **VDataLength**.

WriteData

Write fixed length data record.

SHORT NAME WRTREC

TYPE Low level data file function

DECLARATION

```
COUNT WriteData(datno,recbyt,recptr)
COUNT datno;
LONG recbyt;
VOID *recptr;
```

DESCRIPTION **WriteData** writes fixed length record buffer pointed to by *recptr* into record position *recbyt* of data file *datno*.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful write.
29	ZREC_ERR	*recbyt* is zero.
30	LEOF_ERR	*recbyt* exceeds the logical end-of-file maintained in the data file header.
33	DNUL_ERR	*recptr* is null.
35	SEEK_ERR	lseek failed while preparing for read.
37	WRITE_ERR	Operating system could not execute write. Most likely cause is a full disk or directory.
57	DADV_ERR	Proper lock not found by the FairCom Server. See *ctsrvr.c*

EXAMPLE

```
COUNT  datno;
TEXT recptr[1024];

if (WriteData(datno,NewData(datno),recptr))
   printf("\nCould not write record. Error #%d",uerr_cod);
```

SEE ALSO **ReadData**

WriteVData

Write variable length data record.

SHORT NAME WRTVREC

TYPE Low level data file function

DECLARATION

```
COUNT WriteVData(datno,recbyt,recptr,varlen)
COUNT   datno;
LONG recbyt;
VOID   *recptr;
VRLEN  varlen;
```

DESCRIPTION **WriteVData** writes *varlen* bytes from record buffer pointed to by *recptr* into record position *recbyt* of data file *datno*.

RETURN

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful write.
33	DNUL_ERR	*recptr* is null.
35	SEEK_ERR	lseek failed while preparing for read.
37	WRITE_ERR	Operating system could not execute write. Most likely cause is a full disk or directory.
48	FMOD_ERR	*datno* is not assigned to variable length file.
57	DADV_ERR	Proper lock not found by the FairCom Server. See *ctsrvr.c*
148	VLEN_ERR	*varlen* bytes will not fit into file at position *recbyt*.
159	VPNT_ERR	*recbyt* is zero.

EXAMPLE

```
LONG          recbyt;
COUNT         datno;
TEXT          recptr[1024];

scanf("%1023s",recptr);
recbyt = NewVData(datno,strlen(recptr));
if (WriteVData(datno,recbyt,recptr,strlen(recptr)))
   printf("\nCould not write record. Error #%d",
          uerr_cod);
```

SQL Functions/Statements

This section lists the SQL statements and the C Interface functions used to access the FairCom SQL Server.

2. SELECT Statement

The SELECT statement is the primary command that you will use to access and display information in the database. There are many variations of this statement, which we will discuss in this chapter.

Syntax

SELECT [DISTINCT | ALL] *field_expression* { , *field_expression* }*
FROM *table_spec* { , *table_spec* }*
[WHERE *search_condition*] ;

The WHERE clause is optional, and there are additional possible clauses (GROUP BY, HAVING, ORDER BY, and COMPUTE).

The following sections begin with the simplest SELECT statement, and illustrate SELECT capabilities in progressively complex examples.

Simple SELECT

The simplest SELECT statement is:

```
SELECT * FROM s;
```

The asterisk (*) means all fields. This statement prints all records in *s*.

Instead of using *, one or more field names can be used. For example:

```
SELECT pno FROM sp;
SELECT pno,qty FROM sp;
```

The Authorization Identifier Prefix

In a query, you may used *authid.tablename* in place of *tablename*. For example, instead of

```
SELECT * FROM s;
```

you can use

```
SELECT * FROM john.s;
```

If the current authorization identifier has SELECT privileges for table *s*, then the SELECT is allowed.

If the authorization identifier prefix is not specified, the current authorization identifier is used.

SELECT with Condition

To limit the retrieved data to certain records a condition or WHERE clause is used. For example:

```
SELECT sno FROM s WHERE city = 'Paris';
```

The equals operator (=) means "is equal to". Other comparison operators are:

- Not equal <>
- Less than <
- Less than or equal <=
- Greater than >
- Greater than or equal >=

More than one comparison can be used with the AND and OR operators. For example:

```
SELECT sno FROM s  WHERE city='Paris' AND status=20;
```

SELECT with DISTINCT - Eliminating Duplicate Rows

The keyword DISTINCT is used to eliminate duplicate rows in the retrieved data. For example:

```
SELECT DISTINCT pno FROM sp;
```

The opposite of DISTINCT is ALL;

```
SELECT ALL pnoFROM sp;
```

Because ALL is the default, this is equivalent to;

```
SELECT pno FROM sp;
```

SELECT with Two Tables

More than one table can be accessed with a SELECT statement. This is sometimes referred to as the ability to 'join' tables. In FairCom SQL Server, two (or more) files may be joined, and a file may also be joined to itself (an example follows). For example, the following query can be described in words as "for each part supplied, get part number and names of all cities supplying the part".

```
SELECT DISTINCT pno,city FROM sp,s  WHERE sp.sno=s.sno;
```

Two tables are used, *s* and *sp*. Since both tables have a field named *sno*, the table name is used as a prefix to specify which *sno* we are referring to. Thus, *sp.sno* refers to the *sno* column associated with table *sp*, while *s.sno* refers to the *sno* column associated with table *s*.

The condition *sp.sno = s.sno* tells FairCom SQL Server how to join files *sp* and *s*. Note that the table which appears first in the query, *sp*, becomes the master file, and records from *s* are retrieved which match *sp.sno*. The FairCom SQL Server may, for optimization, reverse the master and slave designations.

The following example joins file *s* to itself. The operation can be expressed in words as "Get all pairs of supplier numbers such that the two suppliers are located in the same city".

```
SELECT first.sno,second.sno FROM s first, s second
    WHERE first.city = second.city AND first.sno < second.sno;
```

The files are specified as *s first* and *s second*. *first* and *second* are aliases (or correlation names) for *s*. The first 'incarnation' of *s* is subsequently referred to as *first*, and the second 'incarnation' is subsequently referred to as *second*.

SELECT with ANY

Consider the following two queries, which both produce the same list. The operation is 'Get supplier names for suppliers who supply part P2'.

```
SELECT DISTINCT sname FROM s,sp WHERE s.sno = sp.sno AND
    sp.pno = 'P2';

SELECT sname FROM s WHERE sno  =  ANY
SELECT sno FROM sp WHERE pno = 'P2';
```

The second form introduces the ANY keyword, which can be used after =, <, < =, >, > =, and < >.

The format here is

```
SELECT sname FROM s WHERE sno  =  ANY (sub_query);
```

where sub-query is an entire additional select statement. The result of the sub-query is effectively a list of values. The FairCom SQL Server then tests whether *sno* is equal to ANY of the values retrieved by the sub-query. If *sno* is equal to at least one of the values retrieved by the sub-query, then = ANY is true.

Here is another example, using < ANY.

```
SELECT sno FROM s WHERE status < ANY (SELECT status FROM s);
```

This finds values of *sno* for suppliers whose status is less than the status of at least one other supplier.

VALUE List

Wherever a sub-query is allowed, it may be replaced by a list of values surrounded by parentheses. For example:

```
SELECT sname FROM s WHERE sno = ANY ('Smith', 'Jones');
```

SELECT with IN

IN is used with sub-queries and is the same as = ANY. For example:

```
SELECT sname FROM s WHERE sno IN
(SELECT sno FROM sp WHERE pno = 'P2');
```

SELECT with Three Levels

A sub-query can reference another sub-query. For example:

```
SELECT sname FROM s WHERE sno IN
(SELECT sno FROM sp WHERE pno IN
(SELECT pno FROM p WHERE color = 'Red'));
```

The number of levels allowed in a SELECT statement is, by default, 3. This can be increased by assigning a value higher than three to global variable **_maxcsr** in an application program.

SELECT with Interblock Reference

An interblock reference occurs when a sub-query uses a value (in its search condition) of a column from an outer query. For example, this query is a variation on a previous example:

```
SELECT sname FROM s WHERE 'P2' IN
(SELECT pno FROM sp WHERE sno = s.sno);
```

Incidentally, this form is may be less efficient because of the interblock reference. However, there are many queries which cannot be performed without an interblock reference.

The significance of the interblock reference is that the sub-query must be performed for *each* row in the outer query. Since the value from the outer query changes, the inner query results are different for each outer query row.

Interblock Reference - Same Table in Both SELECTS

In this query, the interblock reference is to two incarnations of the same table.

```
SELECT DISTINCT pno FROM sp spx WHERE pno IN
(SELECT pno FROM sp WHERE sno < > spx.sno);
```

This may be phrased 'Gets part numbers for all parts supplied by more than one supplier'. For each part (each outer query row), is there another supplier of that part (is the sub-query empty)? This type of query cannot be performed without the interblock reference.

SELECT with ALL

The ALL keyword is used like the ANY keyword, and can follow the =, < >, <, < =, >, and > = operators. The following query 'gets supplier names for suppliers who do not supply part P2'.

```
SELECT sname FROM s WHERE 'P2' < > ALL
(SELECT pno FROM sp WHERE sno = s.sno);
```

Note again the use of the interblock reference.

SELECT with Comparison Operator and Sub-Query

The comparison operators (=, < >, <, < =, >, > =) can reference a sub-query without ALL or ANY. For example, the following query 'gets supplier numbers for suppliers who are located in the same city as supplier S1'.

```
SELECT sno FROM s WHERE city =
(SELECT city FROM s WHERE sno = 'S1');
```

Note that sub-queries in this context must return exactly one value. If zero values or two or more values are returned, an error occurs.

SELECT with EXISTS

EXISTS is used with sub-queries to determine whether any values exist for a particular condition. The following query is another example of 'get supplier names for suppliers who supply part P2'.

```
SELECT sname FROM s WHERE EXISTS
(SELECT * FROM sp WHERE sno = s.sno AND pno = 'P2');
```

Sub-queries associated with EXISTS usually use SELECT *, since all that matters is whether any rows are found by the sub-query. The meaning of EXISTS can be

reversed with the keyword NOT. For example, the following query 'gets supplier names for suppliers who do not supply part P2'.

```
SELECT sname FROM s WHERE NOT EXISTS
(SELECT * FROM sp WHERE sno = s.sno AND pno = 'P2');
```

Here is a three level query using NOT EXISTS. The query 'gets supplier names for suppliers who supply all parts'.

```
SELECT sname FROM s WHERE NOT EXISTS
(SELECT * FROM p WHERE NOT EXISTS
(SELECT * FROM sp WHERE sno = s.sno AND pno = p.pno));
```

To understand how this works, the following is an alternate word expression of the query: 'select supplier names for suppliers such that there does not exist a part that they do not supply'. Note that the innermost sub-query is correlated to both the outermost query and the middle query.

SELECT with Computed Values

The field specifications in a SELECT statement may be expressions rather than simple field names. For example:

```
SELECT pno, WEIGHT*454 FROM p;
```

Supported operators are *, /, +, -, and parentheses.

SELECT with ANSI Functions

There are five ANSI functions: SUM, AVG, MIN, MAX, and COUNT. These functions are called "aggregate" functions because they make sense only when applied to more than one row. Aggregate functions can be used in the SELECT list, or in a HAVING clause (described later). Aggregate functions cannot be used in a WHERE clause.

Format: *ANSI_function (expression)* COUNT (*)

NOTE: COUNT counts the number of rows produced by a SELECT. COUNT is the only function which may be used with *. DISTINCT can be used to count the number of non-identical rows.

ANSI Function	Function Type	*expression* Type
COUNT	DECIMAL	any type
SUM	DECIMAL or DOUBLE PRECISION	SMALLINT, INTEGER, DECIMAL, FLOAT or DOUBLE PRECISION
AVG	DECIMAL or DOUBLE PRECISION	SMALLINT, INTEGER, DECIMAL, FLOAT or DOUBLE PRECISION
MIN	DECIMAL or DOUBLE PRECISION	SMALLINT, INTEGER, DECIMAL, FLOAT or DOUBLE PRECISION
MAX	DECIMAL or DOUBLE PRECISION	SMALLINT, INTEGER, DECIMAL, FLOAT or DOUBLE PRECISION

Examples

```
SELECT COUNT(*) FROM s;
```

Counts the number of rows in *s*.

```
SELECT COUNT(*) FROM sp WHERE pno = 'P2';
```

Counts the number of rows in *sp* whose *pno* value is 'P2'.

```
SELECT COUNT(DISTINCT pno FROM sp;
SELECT qty, SUM(qty) FROM sp WHERE pno = 'P2';
```

Lists each *qty* and computes the sum of the *qtys* for records in *sp* whose part number is 'P2'.

```
SELECT SUM(qty) FROM sp WHERE pno = 'P2';
```

Like the preceding query, except that only one line is retrieved.

```
SELECT AVG(qty) FROM sp;
```

Prints the average of all the *qty* values in *sp*.

```
SELECT AVG(qty) FROM sp WHERE pno = 'P2';
```

Prints the average of *qty* values whose *pno* value is 'P2'.

```
SELECT MIN(qty) FROM sp;
```

Prints the lowest *qty* value in *sp*.

```
SELECT MIN(qty) FROM sp WHERE pno = 'P2';
```

Prints the lowest *qty* value from rows in *sp* whose *pno* value is 'P2'.

SELECT MAX(*qty*) FROM *sp*;

Prints the highest *qty* value in *sp*.

SELECT MAX(*qty*) FROM *sp* WHERE *pno* = 'P1';

Prints the highest *qty* value from rows in *sp* whose *pno* value is 'P1'.

SELECT with String Functions

Description

In addition to the ANSI SQL functions described in the preceding section, the FairCom SQL Server provides a number of functions which are not part of ANSI SQL. These functions can be used in a WHERE clause, because they operate on individual values. In the descriptions which follow, *char_exp* means "character expression", and *num_exp* means "numeric expression".

SELECT with LENGTH Function

Format: LENGTH (*char_exp*)

Types:	LENGTH	DECIMAL
	char_exp	CHAR

Example

SELECT LENGTH(*sname*) FROM *s*;

LENGTH takes a CHAR column and produces the length of each entry for that column. The resulting type is DECIMAL.

SELECT with RTRIM Function

Format:RTRIM (*char_exp1*, *char_exp2*)

Types:	RTRIM	CHAR
	char_exp1	CHAR
	char_exp2	CHAR

RTRIM takes a CHAR column and removes characters from the end. More than one character can be specified within the quotes.

Example

SELECT RTRIM(*sname*, ' ') FROM *s*;

Removes trailing spaces from *sname*.

SELECT with RPAD Function

Format: RPAD (*char_exp1*, *char_exp2*, *num_exp3*)

Types:	RPAD	CHAR
	char_exp1	CHAR
	char_exp2	SMALLINT, INTEGER, or DECIMAL

	char_exp3	CHAR, length must be 1

RPAD adds characters to the end of a CHAR column. The second argument specifies how may characters to add. In the example, three dots are added to *sname*. For RPAD, only one character is allowed within the quotes.

Example SELECT RPAD(*sname*, 3, '.') FROM *s*;

SELECT with LOWER Function

Format:	LOWER (*char_exp*)	
Types:	LOWER	CHAR
	char_exp	CHAR

Example SELECT * FROM *s* WHERE LOWER(*city*) = 'paris';

LOWER forces all characters of a CHAR column to lower case. This is useful for comparisons when matches are desired regardless of case. In the example, the WHERE condition is satisfied for values PARIS, Paris, paris, etc.

SELECT with UPPER Function

Format:	UPPER (*char_exp*)	
Types:	UPPER	CHAR
	char_exp	CHAR

Example SELECT * FROM *s* WHERE UPPER(*city*) = 'PARIS';

Similar to LOWER except that upper case is used. The WHERE condition matches PARIS, Paris, paris, etc.

SELECT with SUBSTR Function

Format:	SUBSTR (*char_exp1*, *num_exp2*, *num_exp3*)	
Types:	SUBSTR	CHAR
	char_exp1	CHAR
	num_exp2	SMALLINT, INTEGER, or DECIMAL
	num_exp3	SMALLINT, INTEGER, or DECIMAL

Example SELECT *city*, SUBSTR(*city*, 1, 2) FROM *s*;

SUBSTR extracts a portion of a CHAR column. The second parameter specifies the first character, and the third parameter specifies the length. If the second parameter specifies a starting locating past the end of the column, then SUBSTR returns ' ' (blank).

SELECT with CONCAT Function

Format: CONCAT (*char_exp1*, *char_exp2*)
Types:

CONCAT	CHAR
char_exp1	CHAR
char_exp2	CHAR

CONCAT combines CHAR columns or expressions.

Example SELECT CONCAT(*sname*, CONCAT(' ', *city*)) FROM *s*;

SELECT with DATE/TIME FUNCTIONS

The FairCom SQL Server provides a series of DATE/TIME functions which allow manipulations of DATE/TIME data. Both the DATE/TIME type and the associated functions are extensions and are not part of ANSI SQL. DATE/TIME functions operate on individual values, and can be used in the WHERE clause.

For the DATE/TIME function examples, we use a table called **DT**. It has two fields, **F1** and **F2**. **F1** is type CHAR, and identifies the row. **F2** is type DATE, and contains DATE/TIME values. *date_exp* means "date/time expression", *char_exp* means "character expression", and *num_exp* means "numeric expression".

SELECT with TRUNCATE Function

Format: TRUNCATE (*date_exp*)
Types:

TRUNCATE	DATE/TIME
date_exp	DATE/TIME

TRUNCATE sets the time portion of a DATE/TIME field to 0.

Example SELECT *f1*, TRUNCATE(*f2*) FROM *dt*;

SELECT with TO_DATE Function

Format: TO_DATE (*date_exp1*, *char_exp2*)
Types:

TO_DATE	DATE/TIME
date_exp1	CHAR
char_exp2	CHAR (pattern string)

The TO_DATE function is included mostly because other SQL implementations require it, and some users are accustomed to using it. FairCom SQL Server normally specifies a data as [DDMMYYYY]. TO_DATE can be used to convert dates in other formats. See also the SET DATE FORMAT command.

Example TO_DATE(*field_name*, 'MMDDYYYY')

TO_DATE may be useful when used in conjunction with string manipulation functions.

SELECT with TO_CHAR Function

Format: TO_CHAR (*date_exp1*, *char_exp2*)
Types:

TO_CHAR	CHAR
date_exp1	DATE/TIME
char_exp2	CHAR (pattern string)

TO_CHAR is the reverse of TO_DATE, and converts a DATE/TIME field to CHAR form. It may be useful in conjunction with TO_DATE and string manipulation functions.

Example TO_CHAR(*field_name*, 'MM-DD-YYYY')

SELECT with Time Addition

Format: *add_date_time* (*date_exp1*, *num_exp2*)
Types:

add_date_time	*Value from table below*
date_exp1	DATE/TIME
num_exp2	SMALLINT, INTEGER, or DECIMAL.

This form of the SELECT statement will add the numeric value *num_exp2* to the date/time value *date_exp1*. The amount of time to be added is specified by *add_date_time*, which is a value from the table below.

add_date_time	**value added**
ADD_YEARS	years
ADD_MONTHS	months
ADD_DAYS	days
ADD_HOURS	hours
ADD_MINUTES	minutes
ADD_SECONDS	seconds

SELECT with Time Difference

Format: *diff_date_time* (*date_exp1*, *date_exp2*)
Types:

diff_date_time	DECIMAL
date_exp1	DATE/TIME

date_exp2 DATE/TIME

Returns the difference between *date_exp1* and *date_exp2*. in the units specified. The result is negative if *date_exp2* is later than *date_exp1*.

diff_date_time	**unit of difference**	**comment**
YEARS_BETWEEN	Years	
MONTHS_BETWEEN	Months	Years are converted to 12 months
DAYS_BETWEEN	Days	Years are converted to 365 days (366 for leap year), Months are converted to the correct number of days depending on the month (and if it is leap year). if *date_exp2* is later than *date_exp1*.
HOURS_BETWEEN	Hours	Years, months and days are converted to the appropriate number of hours.
MINUTES_BETWEEN	Minutes	Years, months, days and hours are converted to the appropriate number of minutes.
SECONDS_BETWEEN	Seconds	Years, months, days, hours and minutes are converted to the appropriate number of seconds.

SELECT with DATE_REMAINDER

Format: DATE_REMAINDER (*date_exp1*, *date_exp2*, *char_exp3*)
Types: DATE_REMAINDER DECIMAL
date_exp1 DATE/TIME
date_exp2 DATE/TIME
char_exp3 CHAR

This function is used to retrieve the years, months, days, hours, minutes, or seconds which would result from a subtraction of two dates. Which operation is performed depends on the value of *char_exp3*. The following table shows the DATE_REMAINDER result for the various values of *char_exp3* for the following two dates:

date1: 12-11-1990-10:09:08
date2: 01-01-1989-01:01:01

char_exp3	**Function Operation**	**Output for Example**
'Y'	Years	1
'MO'	Months	11

char_exp3	**Function Operation**	**Output for Example**
'D'	Days	10
'H'	Hours	9
'MI'	Minutes	8
'S'	Seconds	7

SELECT with BETWEEN

The BETWEEN operator provides a natural syntax for a condition which is greater than one value and less than another. For example,

```
SELECT * FROM s WHERE status BETWEEN 15 AND 25;
```

This is identical in meaning to

```
SELECT * FROM s WHERE status >= 15 AND status <= 25;
```

SELECT with LIKE

The LIKE operator provides pattern matching capability for strings. The pattern is specified in a manner similar to UNIX regular expressions, except that the symbols are different. For those familiar with Unix regular expressions, SQL uses **%** instead of *****, and **_** instead of the period. For those unfamiliar with UNIX regular expressions:

Character	**Description**
_ (underscore)	Any character (exactly one)
%	Any sequence of characters (zero or more)
others	Themselves

Thus, **%s** matches any string which ends is **s**. **abc** matches only the string **abc**. **a_c** matches any three character string which begins with **a** and ends with **c**, such as **abc**, **azc**, etc. **a%c** matches any string (of whatever length) which begins with **a** and ends with **c**, for example, **abc**, **abdefghijklmnopqrc**, **ac**, etc.

Here is an example of an SQL statement using LIKE:

```
SELECT * FROM s WHERE sname LIKE '%k';
```

This retrieves data for **Clark**, which ends in **k**.

SELECT with GROUP BY

GROUP BY allows records in a file to be grouped, usually for sub-totals but also for display purposes. For example, 'for each part supplied, get the part number and the total quantity supplied of that part'.

```
SELECT pno,SUM(qty) FROM sp GROUP BY pno;
```

The **GROUP BY** clause causes a sub-total to be taken and a line printed whenever the value of *pno* changes.

If there is no explicit **ORDER BY** clause, then the query is ordered by the **GROUP BY** fields.

SELECT with GROUP BY and HAVING

The **HAVING** clause is a conditional expression used to eliminate some groups from a report. For example, 'get part numbers for all parts supplied by more than one supplier'.

```
SELECT pno FROM sp
    GROUP BY pno HAVING COUNT(*) > 1;
```

The function within the **HAVING** clause applies to the group. Thus within a **HAVING** clause **SUM**(*qty*) > 100 may be true for a group although the individual values of *qty* may be less than 100.

SELECT with ORDER BY

ANSI ORDER BY Features

In combination with the features described previously, the order of the retrieved data may be specified with an ORDER BY clause. The format is:

```
ORDER BY order_specification { , order_specification } *
```

where *order_specification* is

```
{ column_name | column_number }
[ASC | ASCENDING | DESC | DESCENDING ]
```

ASC or ASCENDING is the default. For example;

```
SELECT * FROM s ORDER BY status;

SELECT sno, status FROM s WHERE CITY = 'Paris'
ORDER BY status DESC, sno
```

The least example could have been written

```
. . . ORDER BY status DESC, sno ASC
```

or

```
. . . ORDER BY status DESCENDING, sno ASCENDING
```

etc.

In place of the column name, a column number may be used. For example;

```
SELECT sno, status FROM s ORDER BY 2 DESC, 1 ASC
```

Which could also be written

```
. . . ORDER BY status DESC, 1
```

or

```
. . . ORDER BY 2 DESC, sno
```

etc.

The column number is optional with this type of query. The column number must be used when using ORDER BY with the UNION, INTERSECTION, or MINUS operators, described later.

ORDER BY with NODUP

NODUP prints blanks in a column if the value to be printed is the same as the value for the preceding line. This is often desirable when ordering by a column for which there are multiple report lines for each column value. The syntax is:

```
SELECT ... ORDER BY column_name NODUP ...
```

For example,

```
SELECT * FROM s ORDER BY city NODUP;
```

ORDER BY with BREAK

The BREAK keyword allows the user to skip lines between groups of a report. The syntax is:

```
SELECT ... ORDER BY column_name BREAK n ...
```

where **n** is the number of lines. For example,

```
SELECT * FROM s ORDER BY city BREAK;
```

or

```
SELECT sno, sname, city FROM s
ORDER BY city DESC BREAK 2;
```

To advance the report to the next page, use **BREAK PAGE**, for example:

```
SELECT * FROM s ORDER BY city BREAK PAGE;
```

ORDER BY with COMPUTE

COMPUTE is used to print some value or computation when an ordering field changes. This is most frequently used for printing subtotals. The syntax is:

```
SELECT ... ORDER BY column
COMPUTE value_expression AT column_number
  {, COMPUTE value_expression AT column_number} *
```

For example,

```
SELECT sno, sname, status, city FROM s
ORDER BY city COMPUTE SUM(status) AT 3;
```

Note that the aggregate functions (SUM, AVG, MIN, and MAX) are allowed in the COMPUTE statement.

COMPUTE can be combined with BREAK or BREAK PAGE. For example:

```
SELECT sno, sname, status, city FROM s
ORDER BY city BREAK 2
COMPUTE SUM(status) AT 3 BREAK PAGE;
```

SELECT with AT END REPORT

The AT END REPORT clause is used to print values or calculations at the end of a report. The syntax is similar to the COMPUTE clause. AT END REPORT is

often used to print totals (or grand totals). Aggregate functions are allowed in the AT END REPORT clause. The syntax is:

```
SELECT ... AT END REPORT [ BREAK [ n | PAGE ]]
COMPUTE value_expression AT column_number
  {, COMPUTE value_expression AT column_number} *
```

For example,

```
SELECT * FROM s
AT END REPORT COMPUTE SUM(status) AT 3;
```

Some variations:

```
SELECT * FROM s
AT END REPORT BREAK 2 COMPUTE SUM(status) AT 3;
SELECT * FROM s WHERE status  10
AT END REPORT BREAK PAGE
COMPUTE SUM(status) AT 3;
```

Report Generation Enhancements

This section describes FairCom SQL Server features which allow control of the format of a report produced by a SELECT statement. All the features described are extensions to ANSI SQL and may not be available on other SQL systems.

Output Formatting

As an extension to ANSI SQL, the FairCom Server provides two features that provide you with greater control over the appearance of your output: **pattern strings** and **template strings**.

Pattern String

Each column has a **pattern string** associated with it. This pattern string, stored in the field dictionary, controls the format of that column in select statements.

The FairCom SQL Server allows the default pattern string to be replaced by a pattern string explicitly included in a SELECT statement. The syntax is:

```
SELECT column_name % pattern_string ...
```

For example;

```
SELECT sname % 'XXXXXXXX' FROM s;
```

X allows any character, and the width is 8 because there are 8 **X**'s.

If there are a number of pattern characters that are repeated, as in our example, you can prefix the pattern character with a number to repeat them. For example, instead of 'XXXXXXXX', we could use '8X'.

The pattern characters are:

Pattern Character	Description
X	Any character.
#	Numeric digit or '-'.
L	'Y', 'y', 'N', 'n'.
A	ASCII characters only (rejects values below hex 20 or above hex 7f).
U	Upper case characters only (rejects values below hex 20 or above hex 60).
F	Floating point digit
.	Floating point decimal position
Z	Floating point digit. Same as 'F', except that values are rounded instead of truncated.
YYYY	Four digit year, e.g., 1989.
YY	Two digit year, e.g.
MM	Two digit month, e.g.
MONTH	Full name of month, e.g., DECEMBER.
MMM	Three letter abbreviation of month, e.g., DEC.
DD	Day of month, e.g., 31.
DDD	Day of year, e.g., 250.
D	One digit day of week, e.g., 7.
DAY	Full name of day of week, e.g., MONDAY.
DY	Three letter abbreviation of day, one of MON, TUE, WED, THU, FRI, SAT, SUN.
HH	Hours, e.g., 06
MM	Minutes, e.g., 45
SS	Seconds, e.g., 10.

In the pattern string, there are certain characters designated as **skip characters**. These characters are displayed in the output, but do not consume a character of input. For example, suppose the contents of a CHAR column is:

```
1234567890
```

and the format string is:

```
###-###-####
```

then the output string is:

```
123-456-7890
```

The skip characters are:

Skip Character	Description
-	dash
(	open parenthesis
)	close parenthesis
{	open curly bracket
}	close curly bracket
[	open square bracket
]	close square bracket
/	slash
<	less than
>	greater than
.	period
:	colon
;	semi colon
	space

Template String

The FairCom SQL Server uses a "template string" in combination with the pattern string to provide more control over the output. The template string allows the user to place characters in the output that do not consume any input characters but are not permanent skip characters. When the FairCom SQL Server sees a skip charac-

ter in the pattern string, the character from the corresponding location in the template string is actually printed. If the user does not provide a template string, the pattern string is used as the template string (and the skip character itself is printed). An example might be:

```
column contents:    1234567890
pattern string:     /////###/###/####
template string:    ss#:###-###-####
output string:      ss#: 123-456-7890
```

The syntax for specifying the template string is:

SELECT *value_expression*
    % '*pattern_string*' TP '*template_string*'

NODUP Modifier

The user can follow a column_specification with the keyword NODUP, which prints a blank if the value of a column is the same as the value of that column on the preceding line. The syntax is:

SELECT *column_name* NODUP ...

For example,

SELECT *sno*, *sname*, *city* NODUP FROM *s*;

Column Heading Override

In the FairCom SQL Server, each column has a column heading associated with it. This column heading is stored in the field dictionary. When a table is created, the column heading is set to the field name.

The FairCom SQL Server allows the default column heading to be replaced by a column heading explicitly included in a SELECT statement. The syntax is:

SELECT *column_name column_heading* ...

for example,

SELECT *sname* 'Supplier Name' FROM *s*;

The symbol | has a special meaning within column headings. It is used to split a heading onto several lines. The preceding example with a two line heading is:

SELECT *sname* 'Supplier|Name' FROM *s*;

Column Justification

By default, all columns are left justified in a SELECT statement. The FairCom SQL Server allows the user to specify centering or right justification as part of the SELECT. The syntax is:

```
SELECT column_name justification_specifier ...
```

where *justification_specifier* is **CENTER** or **RIGHT**. For example,

```
SELECT sname CENTER, status RIGHT FROM s;
```

Wrapping

For CHAR columns or expressions, the FairCom SQL Server normally truncates the output if the width is greater than the column width (as specified by the pattern string). The user can specify wrapping or word wrapping in the SELECT statement. The syntax is:

```
SELECT column_name wrap_specifier ...
```

where wrap_specifier is **WRAPPED** or **WORD_WRAPPED**. For example,

```
SELECT CITY % 'XXX' WRAPPED FROM s;
```

Note the use of the pattern string override. Since the default pattern string for CITY specifies the maximum width allowed for CITY, wrapping won't occur unless the default pattern string is overridden (unless CITY is a variable length character field).

Summary of Column Specification Features

The syntax of the SELECT column specification is:

value_expression [*%'pattern_override'* [TP *'template_override'*]]
'column_heading' [LEFT | CENTER | RIGHT]
[WRAPPED | WORD_WRAPPED] [NODUP]

The SELECT INTO Statement

```
SELECT INTO target_specification  { , target_specification }*
...
```

The format of the balance of the statement is the same as the ordinary SELECT.

This retrieves one row in FETCH format using a SELECT statement.

The SELECT INTO statement returns data in the same format as FETCH. The SELECT INTO operation is equivalent to the following sequence: DECLARE CURSOR, OPEN, FETCH, CLOSE. Note that the ANSI specification requires that SELECT INTO return only one record.

3. SQL / C Interface Function Reference

This chapter defines the C language interface functions that you will use to make embedded SQL calls to the database.

destbuf

Memory Buffer Output -Redirects FairCom SQL Server output to a memory buffer.

DECLARATION

```
void  destbuf(addr)
char  *addr;
```

DESCRIPTION

destbuf places FairCom SQL Server in buffer output mode. *addr* specifies the address of the buffer. The user is responsible for allocating space for this buffer and making sure the buffer is large enough to hold the output of any FairCom SQL Server statement which is to be executed. All output is redirected to this buffer.

destbuf is usually used to redirect the results of a FETCH statement.

destfil

Canceling Redirection -Redirects FairCom SQL Server output from buffer to file.

DECLARATION

```
void  destfil()
```

DESCRIPTION

destfil cancels the redirection performed by **destbuf**. FairCom SQL Server is placed in file output mode. Output is directed to the current output file pointer. By default, the output file is standard output. The output file is changed using the **setdest** function.

This only applies to SELECT statements (Not SELECT INTO) used in C programs.

r_fetch

Special version of FETCH.

DECLARATION

```
int   r_fetch(cursorname, buf1, buf2, ...,
                  bufn, ((void*)0))
char  *cursorname;
void  *buf1, *buf2, ..., *bufn;
```

DESCRIPTION

r_fetch is a special form of the FETCH statement. Instead of returning field values using the VARSPEC structure, **r_fetch** returns complete table records using a series of table buffer pointers. There are as many table buffer pointers as there are table references in the named cursor. Following the last table buffer pointer, a NULL pointer marks the end of the buffer pointer list.

Note the following about **r_fetch**:

- Raw record information is returned. Any calculations specified in the select list will not be reflected in the returned data. Calculations in other clauses function normally.
- If the number of pointers does not match the number of tables in the referenced cursor, an error occurs.
- The pointers provided to **r_fetch** are pointers to structures whose members match the columns of the corresponding table. For example, if the cursor references one table, S, then one pointer is provided to **r_fetch**; this is a pointer to a structure with the same format as S. Remember that the dictionary (create) size of char columns is one less than the size to use in a C structure (because of the trailing 0 byte).

r_insert

Insert data into tables.

DECLARATION

```
int   r_insert(cursorname, buf1, ((void*)0))
char  *cursorname;
void  *buf1;
```

DESCRIPTION

r_insert is used to insert data into tables more efficiently than is possible using INSERT INTO. The cursor must be open. *buf1* is usually a pointer to a structure with the appropriate layout.

NOTES

- Use **r_insert** only with cursors which reference only one table.

r_update

Update with record buffer.

DECLARATION

```
int   r_update(cursorname, buf1, ((void*)0))
char  *cursorname;
void  *buf1;
```

DESCRIPTION

r_update allows updating using record buffers. This is convenient when a program retrieves a record using **r_fetch**, modifies that record, and wants to write the modified record into the table. The updated row is the last row retrieved by a FETCH statement or **r_fetch** call.

setdest

Redirecting File Output -Directs FairCom SQL Server output to a file or device.

DESCRIPTION

```
int   setdest(filename)
char  *filename;
```

DESCRIPTION

setdest is used to redirect file output. By default, FairCom SQL Server output is directed to standard output and FairCom SQL Server is in file output mode.

RETURN

If the function worked correctly it will return a value of Zero (NO_ERROR). Any non-zero value means that the opening of *filename* failed.

setvars

setvars initializes structure for FETCH data retrieval.

DESCRIPTION

```
int  setvars(ptr, var, type, len, dpat)
char    *ptr;
VARSPEC *vars;
int      type, len;
char    *dpat;
```

DESCRIPTION **setvars** initializes structure for FETCH data retrieval.

The VARSPEC structure is defined by:

```
typedef struct {
int   type;
int   len;
CIUL  ciul;
char *dtpat;
```

CIUL is a union of pointers, one for each FairCom SQL Server column type.

The user allocates an array of SETVARS pointers. The size of the array is the maximum number of values to be returned by a FETCH statement.

If a user program uses more than one FETCH format, it is best to allocate a separate array for each format.

The **setvars** function is used to initialize the members of the VARSPEC array. Subsequently, the address of the VARSPEC array is provided as an argument to **sql2**.

Parameter *len* is only required for char results. Parameter *dpat* is only required for DATE fields.

SEE ALSO sql2

sql1

Executes FairCom SQL Server statements from C programs.

DESCRIPTION

```
int  sql1(statement)
char   *statement;
```

DESCRIPTION

Executes FairCom SQL Server statements from C programs. Statement is a FairCom SQL Server statement as a character string.

sql1 is used with all FairCom SQL Server statements except those which return data (FETCH and FETCHP).

EXAMPLE

```
retcode = sql1("SELECT * FROM S;");
```

RETURN

The value returned by the function is 0 for no error, or an error code. Further information about an error is available in global variables **err_pos** and **uerr_cod**. The value of **sql_err** is the same as the value returned by **sql1**.

sql2

sql2 executes FETCH statement from within C programs.

DESCRIPTION

```
int  sql2(statement, nv, vars)
char    *statement;
int      nv;
VARSPEC *vars;
```

DESCRIPTION

nv is the number of members of the VARSPEC array (see **setvars**). *statement* is either a FETCH statement. The members of the VARSPEC array must be initialized using **setvars** before the call to **sql2**.

RETURN

The value returned by the function is 0 for no error, or an error code. Further information about an error is available in global variables **err_pos** and **uerr_cod**. The value of **sql_err** is the same as the value returned by **sql2**.

4. SQL Statement Reference

These SQL Statement definitions are presented in alphabetical order, each starting on a new page, to facilitate your search for information. SQL C Interface functions are presented in the prior Chapter. Statement definitions are comprised of the following sections:

- **Type** ANSI or Extension to ANSI
- **Syntax** function syntax according to the rules below
- **Description** detailed description
- **Example** programming example
- **Return** explanation of return values
- **Limitations** restrictions on use and/or shortcomings
- **Extensions** comments on extensions to the ANSI definition
- **See Also** related functions and information

Notational Conventions

The syntax descriptions in the sections which follow use the following conventions:

- Parts of statements which are to be typed exactly as they appear in the text appear in bold typeface, such as **ALTER TABLE**, usually in upper case.
- Curly braces { and } are to group sections of commands which are logically distinct. If followed by *, then the section within the brackets occurs 0 or more times. If followed by +, the section within the brackets occurs 1 or more times.
- Square brackets [and] are used around optional sections.
- Parentheses are used to indicate order of evaluation.
- Descriptions of entries are in italic type in lower case. These entries indicate that the description is replaced by an actual value.
- The vertical bar | means "or".

ALTER TABLE

TYPE Extension

SYNTAX

```
ALTER TABLE table_name
    ADD
    (
    column definition [ BEFORE column_name ]
    { , column definition  [ BEFORE column_name ] }*
    )
DROP ( column_name { , column_name }* )
MODIFY ( column definition { , column definition }* )
```

DESCRIPTION

See CREATE TABLE for the syntax of *column definition*.

ALTER TABLE allows a user to change the structure of a table. New columns can be added with the ADD clause. Existing columns may be modified with the MODIFY clause. Columns may be removed using the DROP clause.

The MODIFY clause is used only to modify the UNIQUE, INDEX, or NOT NULL status of a column. To make more extensive changes to a column, DROP it and ADD it.

The user should only modify the UNIQUE or NOT NULL status of a column if the table is empty. If UNIQUE or NOT NULL status of a column is modified on a non-empty table, an error may occur because duplicate or NULL values of that column may already exist in the table.

A NOT NULL column may only be added to an empty table. When a new column is added to a table, the value NULL is assigned to that column for all existing rows of the table. Thus the ALTER TABLE will fail unless the table is empty.

To use ALTER TABLE, the current authorization identifier must be either the table's owner or have been granted ALL PRIVILEGES for the table.

EXTENSIONS

The ALTER TABLE statement is not part of ANSI SQL.

CLOSE

TYPE ANSI

SYNTAX CLOSE *cursor_name*

DESCRIPTION When a cursor is closed, it is prepared for reopening. After CLOSE **cursor_name**, OPEN **cursor_name** positions the cursor at the beginning of the cursor's table. Executing a COMMIT statement closes all open cursors.

SEE ALSO OPEN

COMMIT

TYPE ANSI

SYNTAX

```
COMMIT WORK;
```

DESCRIPTION In a transaction processing environment, COMMIT finalizes all INSERT, DELETE, and UPDATE activities since the prior COMMIT or ROLLBACK statement, or since the beginning of the session.

EXTENSIONS FairCom SQL Server allows a shorter form of this statement, as:

```
COMMIT;
```

CREATE INDEX

TYPE Extension

SYNTAX CREATE INDEX ON *column_name* FOR *table_name*

DESCRIPTION CREATE INDEX modifies the indexing structure of a table. The existing index is erased and a new index is built.

EXTENSIONS CREATE INDEX is not part of ANSI SQL.

CREATE SCHEMA

TYPE: ANSI

SYNTAX:

CREATE SCHEMA AUTHORIZATION *auth-identifier*
{ CREATE TABLE *statement* }*
{ CREATE VIEW *statement* }*
{ GRANT *statement* }*
{ CREATE INDEX *statement* }*
{ REVOKE *statement* }*
;

DESCRIPTION

FairCom SQL Server supports the ANSI CREATE SCHEMA statement (which includes the CREATE TABLE, CREATE VIEW, and GRANT statements).

The CREATE SCHEMA line is optionally followed by any number of CREATE TABLE, CREATE VIEW, GRANT, CREATE INDEX, or REVOKE statements, followed by a terminating semi-colon. There must be at least one CREATE TABLE, CREATE VIEW, or GRANT statement.

SEE ALSO

AUTHORIZATION, CREATE TABLE, CREATE VIEW, CREATE INDEX, GRANT, REVOKE

EXTENSIONS

ANSI SQL does not allow the form of CREATE without the preceding CREATE SCHEMA clause. However, FairCom SQL Server (as do most SQL products) extends the definition to allow you to drop the CREATE SCHEMA clause. In this case, the current AUTHORIZATION identifier is used.

CREATE TABLE

TYPE Extension (ANSI when used with CREATE SCHEMA)

SYNTAX:

CREATE TABLE *table_name* [READ_ONLY]
(*table element* { , *table element* } {, UNIQUE *column_name*)

DESCRIPTION

This statement is used to define a Table. The Table will be empty, that is it will not contain any Rows.

table element specifies the names and attributes of the Columns in the Table. The general form is:

column_name column_type [NOT NULL] [UNIQUE | INDEX]

column_name will be the name assigned to the column for you to reference in later SQL statements. *column_type* specifies the data type (and size) of the column, and can be one of the following:

CHAR [(*length*)]
VCHAR [(*length*)]
NUMERIC [(*precision* [,*scale*])]
DECIMAL [(*precision* [,*scale*])]
INTEGER
SMALLINT
FLOAT [(precision)]
DOUBLE PRECISION [(*precision*)]
DATE
BOOLEAN

The types are defined as follows:

CHARACTER	Character string
VCHAR	Variable length character string
SMALLINT	16-bit signed integer
INTEGER	32-bit signed integer
DECIMAL	32-bit signed integer with 16 bit multiplier
NUMERIC	Same as DECIMAL
FLOAT	32-bit floating point number.
REAL	Same as FLOAT.
DOUBLE PRECISION	64-bit floating point number.

CHARACTER	Character string
DATE	Structure
BOOLEAN	One bit true/false

unique constraint definition
UNIQUE (column_name { , column_name }*)

NOT NULL means a record cannot be inserted into the file if this particular field is null. **UNIQUE** means that duplicates of this field are not allowed in the data file. Note that UNIQUE can be specified when the column name is declared, or as a separate clause. Specifying UNIQUE causes the field to be indexed. **INDEX** causes FairCom SQL Server to create an index for the field and still allow duplicates.

EXAMPLE

The following CREATE SCHEMA statement creates three tables, called s, p, and sp. These tables are used in examples throughout this manual.

```
CREATE SCHEMA AUTHORIZATION PUBLIC
    CREATE TABLE s (
    sno CHAR(5) NOT NULL,
    sname CHAR(20),
    status DECIMAL(3),
    city CHAR(15),
    UNIQUE (sno))

CREATE TABLE p (
pno CHAR(6) NOT NULL,
pname CHAR(20),
color CHAR(6),
weight DECIMAL(3),
city CHAR(15),
UNIQUE (pno))

CREATE TABLE sp (
sno CHAR(5) NOT NULL,
pno CHAR(6) NOT NULL,
qty DECIMAL(5),
UNIQUE (sno,pno))
;
```

Alternately, each CREATE TABLE can be a separate statement, for example:

```
CREATE TABLE s (
    sno CHAR(5) NOT NULL UNIQUE,
    sname CHAR(20),
    status DECIMAL(5,2),
    city CHAR(15))
```

- In the alternate form of CREATE TABLE, *sno* is made UNIQUE as well as NOT NULL, and the final UNIQUE clause is eliminated. The two forms create exactly the same file. When the short form is used, the current AUTHORIZATION identifier becomes the table's AUTHORIZATION identifier.

EXTENSIONS

- The alternate form is, strictly, ANSI level 1. With the implied authorization identifier, however, ANSI level 2 performance occurs.
- The READ_ONLY keyword is an extension to ANSI SQL. While the ANSI specification requires support of read-only tables, no mechanism for creating a read-only table is defined. Note that it only makes sense to create a read-only table if the data is inserted into the table by some non-SQL process, typically an application program. A more normal procedure is to create a read-only view to protect a base table.
- The VCHAR, DATE and BOOLEAN types are extensions and not part of ANSI SQL.
- The modifier INDEX is not part of ANSI SQL.

SEE ALSO ALTER TABLE, CREATE INDEX, CREATE SCHEMA

CREATE VIEW

TYPE: ANSI

SYNTAX:

```
CREATE VIEW view_name [ READ_ONLY ]
( column_name { , column_name }* )
AS select_statement [ WITH CHECK OPTION ]
```

DESCRIPTION

Creating a view, also called a virtual table, does not actually create a table. It creates a logical entity which relates to actual tables (called "base" tables), and which can be used as if it were a table.

The simplest use of a view is to limit access to some of the columns in a table. For example, suppose you wanted to give a user access to fields *sno* and *sname* from file *s* (as defined in the preceding CREATE TABLE examples), and not allow that user access to *status* and *city*. You could define a view, say *s1*, which references those columns. The user can then use *s1* as if it referred to a real file which contains only those fields. The CREATE VIEW statement might be:

```
CREATE SCHEMA AUTHORIZATION ted
CREATE VIEW s1 AS SELECT sno,sname FROM s;
```

or

```
CREATE VIEW s1 AS SELECT sno,sname FROM s;
```

select_statement is a SELECT statement as described in a another section. Most SELECT features are allowed in view definitions. The capabilities of views will be easily understood when the capabilities of the SELECT statement have been explained.

Some views are updatable, while other views are not. An updatable view can be used with an INSERT INTO, DELETE, or UPDATE statement. If a view is not updatable, then it is treated as if it is a read-only base table.

A view is updatable if:

- The first table name in the SELECT statement is an updatable base table and
- Each column reference in the view specifies a column in that base table.

If WITH CHECK OPTION is used, then rows are not allowed into the base table which conflict with the WHERE clause of the SELECT statement in the view definition. For example, suppose the view definition is:

```
CREATE VIEW v1 AS
    SELECT * FROM s WHERE status = 20
```

```
WITH CHECK OPTION;
```

Then the following INSERT statement is allowed (the third value is *status*):

```
INSERT INTO v1 VALUES ('S6', 'Mr. Ed', 30, 'Wash D.C.')
```

However the following INSERT statement is not allowed because it does not satisfy the WHERE clause:

```
INSERT INTO v1 VALUES ('S6','Mr. Ed', 10, 'Wash D.C.')
```

NOTES

- When the alternate form of CREATE VIEW is used, the current AUTHORIZATION identifier becomes the view's AUTHORIZATION identifier.
- If WITH CHECK OPTION is not specified for a CREATE VIEW, and the view is updatable, then it is possible to insert records into the underlying base table which are not retrieved by the view. This can be confusing, since it may appear that the insert operation did not work correctly.
- View insertions and updates are subject to the UNIQUE and NOT NULL constraints of the underlying base table(s). An insert/update on a view may fail because the value of a UNIQUE field is duplicated, even though this value is not displayed when doing SELECTs on the view (due to the WHERE clause). If the view definition does not include a base table column which is NOT NULL, then insertions will not succeed for that view.

DECLARE CURSOR

TYPE ANSI

SYNTAX

```
DECLARE cursor_name CURSOR FOR select_statement
```

DESCRIPTION

A cursor is an object used in program interface mode to obtain access to rows of a table one at a time. It is created by a DECLARE CURSOR. The DECLARE CURSOR statement includes a SELECT statement which specifies the data to be accessed. After the cursor is declared, it is made available using the OPEN statement. The data is then obtained, one row at a time, using the FETCH statement.

The *select_statement* can use all features of the SELECT statement except those involved with report formatting. The *cursor_name* is used to refer to the cursor in other statements.

The DECLARE CURSOR statement is also available in the interpreter, along with the FETCH, OPEN, and CLOSE statements described below. The FETCH statement produces a sensible output for the screen. This is intended as a convenience to allow the developer to declare a cursor and fetch and inspect the retrieved data, to verify that the cursor works as expected.

Cursors are not saved on disk. Once declared, they exist until the end of the program run.

SEE ALSO FETCH, OPEN, CLOSE

DELETE

TYPE ANSI

SYNTAX

DELETE FROM *table_name*
{WHERE *search_condition*} | {WHERE CURRENT OF *cursor_name*} ;

DELETE FROM deletes one or more rows of a table using either a WHERE clause or a WHERE CURRENT OF clause.

The searched DELETE uses a WHERE clause. It can delete 0 records (if none match the search condition), or one or more records. In the interpreter, FairCom SQL Server displays the number of records deleted.

The positioned delete uses a WHERE CURRENT OF clause to delete one record at the current position of the named cursor.

cursor_name must be opened and must be positioned at a record. The usual sequence to accomplish this is DECLARE CURSOR, OPEN, one or more FETCHs, then the positioned DELETE.

DROP TABLE

TYPE Extension

SYNTAX DROP TABLE *table_name*

DESCRIPTION DROP TABLE eliminates all entries for a table from the FairCom SQL Server dictionaries. At that point, the table no longer exists as far as FairCom SQL Server is concerned. However, FairCom SQL Server does not erase the files associated with the table.

If there are views associated with a table, you cannot drop the table without first dropping the views with DROP VIEW.

EXTENSIONS DROP TABLE is not part of ANSI SQL.

SEE ALSO DROP VIEW.

DROP VIEW

TYPE Extension

SYNTAX DROP VIEW *view_name*

DROP VIEW eliminates all entries for a view from the FairCom SQL Server dictionaries. DROP VIEW has no effect on the base table(s) associated with the view.

EXTENSIONS

- DROP VIEW is not part of ANSI SQL.

FETCH

TYPE ANSI with extension

SYNTAX

FETCH *cursor_name*
[INTO *target_specification* { , *target_specification* }*]

DESCRIPTION

The first FETCH statement after an OPEN retrieves data for the first row of the table associated with *cursor_name*. Subsequent FETCHes retrieve data for additional rows.

The FETCH operation establishes a "current record" for the cursor. This record is available for UPDATE and DELETE operations.

The FETCH INTO form must be used when using the FairCom SQL Server preprocessor.

EXTENSIONS The INTO portion of the FETCH statement is required, not optional, in ANSI SQL.

SEE ALSO OPEN, UPDATE, DELETE

FETCHP

TYPE Extension

SYNTAX

FETCHP *cursor_name*
[INTO *target_specification* { , target_specification }*]

DESCRIPTION

FETCHP is like FETCH, except that the previous record rather than the next record is fetched. FETCHP is not always available for all tables.

> **The FETCHP INTO form must be used when using the FairCom SQL Server preprocessor.**

EXTENSIONS FETCHP is not a part of ANSI SQL

SEE ALSO FETCH

GRANT

TYPE Extension (ANSI when used with CREATE SCHEMA)

SYNTAX GRANT *rights* ON *table_name* TO *auth_id*

DESCRIPTION The GRANT statement allows control of access to tables. Privileges can be used to specify who can insert, update, or delete rows of a table.

The *rights* clause can be one or more of the following:

```
ALL PRIVILEGES
SELECT
INSERT
UPDATE
DELETE
```

Each table has an owner, identified by an AUTHORIZATION identifier. The table's owner automatically has all privileges for a table. The owner is specified during the CREATE statement. If the long form of CREATE is used, the first line of a CREATE is:

```
CREATE SCHEMA AUTHORIZATION authid
```

where *authid* is replaced is an arbitrary name chosen by the user. In the example creates, we use:

```
CREATE SCHEMA AUTHORIZATION PUBLIC
CREATE TABLE s ( ...
```

Table *s* is owned by PUBLIC. PUBLIC is a special name, which matches any AUTHORIZATION identifier.

If the short form of CREATE TABLE is used, then the current AUTHORIZATION identifier becomes the owner of the file.

There are four privileges: SELECT, INSERT, DELETE, and UPDATE. The privileges correspond to the SQL operations that a user is allowed to perform on a given table. When created, a table's owner has all rights, and other users have no rights. If the table is owned by PUBLIC, then all users have all rights.

When FairCom SQL Server is started, the current AUTHORIZATION identifier is set to PUBLIC. The user can change the current AUTHORIZATION identifier using the SET AUTHORIZATION statement, described later.

The GRANT statement allows a user to make his tables accessible to other users. Initially, only the file's owner has privileges. The owner can grant rights to specific

users. Also, the user can grant rights to all users with the special authorization identifier PUBLIC.

For the following discussion, assume that table *s* is owned by *george*.

The following GRANT statement grants all rights for file *s* to user *joe*:

```
CREATE SCHEMA AUTHORIZATION george
GRANT ALL PRIVILEGES ON s TO joe
```

joe has SELECT, INSERT, UPDATE, and DELETE privileges. However, *joe* cannot grant rights to other users. The following GRANT statement grants all rights for file *s* to user *john* and allows *john* to grant rights to other users:

```
CREATE SCHEMA AUTHORIZATION george
GRANT ALL PRIVILEGES ON s TO john
    WITH GRANT OPTION
```

Now either *george* (the table's owner) or *john* can grant rights to other users.

Privileges can be GRANTed to more than one user in a single GRANT statement. For example, the following GRANTs all privileges to users *john* and *joe*:

```
CREATE SCHEMA AUTHORIZATION george
GRANT ALL PRIVILEGES ON s TO john,joe
```

The following GRANT statement allows user *mary* to select data from file *s* and to insert data into file *s*:

```
CREATE SCHEMA AUTHORIZATION george
GRANT SELECT,INSERT ON s TO mary
```

mary cannot perform delete or update operations on *s*.

The following GRANT statement grants the select privilege to all users:

```
CREATE SCHEMA AUTHORIZATION george
GRANT SELECT ON s TO PUBLIC
```

For the update operation only, rights can be granted to update some but not all columns of a table. If no columns are mentioned, then by inference all columns are updatable. The following statement allows user *john* to update columns *sno* and *sname* of file *s*:

```
CREATE SCHEMA AUTHORIZATION george
    GRANT UPDATE (sno,sname) ON s TO john
```

The following statement grants update rights on all columns:

```
CREATE SCHEMA AUTHORIZATION george
    GRANT UPDATE ON s TO john
```

The following statement grants all rights, including update rights on all columns:

```
CREATE SCHEMA AUTHORIZATION george
    GRANT ALL PRIVILEGES ON s TO john
```

EXTENSIONS

GRANT is an extension if not used as a part of a CREATE SCHEMA statement.

INSERT INTO

TYPE ANSI

There are two forms of this command - Standard and With Select;

Standard

```
INSERT INTO table_name
[ ( column_name, { , column_name }* ) ]
VALUES ( value, { , value }* )
```

DESCRIPTION INSERT INTO is used to insert data into a table. The list of column_names following INSERT INTO *table_name* is optional. If the list is included, the number of values must be the same as the number of column_names listed. If the list is omitted, then the number of values must be the same as the number of columns associated with the table.

The value list types must match the types of the columns associated with the table. CHAR fields must be initialized with quoted strings. Numeric columns must be initialized with appropriate numeric data. DATE/TIME columns must be initialized with valid DATE/TIME values. Note that in a value list the DATE/TIME value is entered as if it is a CHAR, for example, '10061989'.

NOTES

- Data is not inserted into the table until COMMIT, and the INSERTed data can be discarded using ROLLBACK.

With Select

Syntax

```
INSERT INTO table_name
[ ( column_name, { , column_name }* ) ]
SELECT value_expression { , value_expression }*
FROM table_name
THERE condition ;
```

DESCRIPTION This form of the INSERT INTO statement inserts data SELECTed from one table into a second table.

The list of *column_names* following INSERT INTO table_name is optional. If the list is included, the number of *value_expressions* in the SELECT statement must be the same as the number of *column_names* listed. If the list is omitted, then the number of *value_expressions* in the SELECT statement must be the same as the number of columns associated with the table.

The *value_expression* types must match the table column types. CHAR fields must be initialized with CHAR *value_expressions*. Numeric columns must be initialized with appropriate numeric data. DATE/TIME columns must be initialized with valid DATE/TIME values. BOOLEAN columns must be initialized with BOOLEAN or numeric types.

NOTES

- Data is not permanently inserted into the table until COMMIT, and the INSERTed data can be discarded using ROLLBACK.

OPEN

TYPE ANSI

SYNTAX OPEN *cursor_name*

DESCRIPTION Access a cursor created in a DECLARE CURSOR statement. *cursor_name* must be declared using DECLARE CURSOR before it can be opened. Once open, the cursor can be accessed by statements which retrieve, insert, delete, or modify data.

OPEN is a separate operation from DECLARE CURSOR because it is sometimes convenient to CLOSE and then OPEN a cursor, to begin data retrieval from the beginning of the table.

SEE ALSO DECLARE CURSOR, CLOSE

REVOKE

TYPE Extension

SYNTAX

REVOKE *rights* ON *table_name* FROM *auth_id*

DESCRIPTION

REVOKE reverses the process performed by GRANT. The table's owner can REVOKE privileges for any authorization id. Other users can REVOKE privileges for those rights they were granted WITH GRANT OPTION. That is, if I use

GRANT UPDATE ON *s* TO *joe* WITH GRANT OPTION

then *joe* can *grant* or REVOKE UPDATE privilege for table *s*. Note that *joe* can revoke his own privileges.

rights can be one or more of the following:

ALL PRIVILEGES
SELECT
INSERT
UPDATE
DELETE

ROLLBACK WORK

TYPE ANSI

SYNTAX

```
ROLLBACK WORK;
```

DESCRIPTION In a transaction processing environment, ROLLBACK cancels all pending INSERT, DELETE, and UPDATE activities since the prior COMMIT or ROLLBACK statement, or since the beginning of the session.

EXTENSIONS FairCom SQL Server allows a shorter form of this statement, as:

```
ROLLBACK;
```

SELECT

TYPE ANSI

SYNTAX

SELECT [DISTINCT] *field_expression* { , *field_expression* }*
FROM *table_spec* { , *table_spec* }*
[WHERE *search_condition*] ;

DESCRIPTION

The SELECT statement is your primary data retrieval and display command.

The WHERE clause is optional, and there are additional possible clauses (GROUP BY, HAVING, ORDER BY, COMPUTE, and USING).

Details on using the SELECT command are found in Chapter 2.

SET AUTHORIZATION

TYPE Extension

SYNTAX SET AUTHORIZATION TO *authorization_identifier*

DESCRIPTION The user can change the current authorization identifier using the SET AUTHORIZATION statement. For example, to change the authorization identifier to *ted*:

SET AUTHORIZATION TO *ted*;

The SET AUTHORIZATION statement should not be used on multi-user systems using a single database server. Instead, the **sql1** and **sql2** functions should be modified to accept an additional argument, the authorization identifier, which should then be copied (string) into global variable *authname*.

UPDATE

TYPE ANSI

SYNTAX

UPDATE *table_name*
SET *field_name* = *expression*, ..., *field_name* = *expression*
{WHERE *search_condition*} | {WHERE CURRENT OF *cursor_name*};

UPDATE is used to update one or more rows of a table based on a WHERE clause. There are two forms of this statement, using the WHERE clause or the WHERE CURRENT OF clause.

In the first form, the searched UPDATE modifies one or more records satisfying a WHERE clause. The values provided must match the type of the table columns involved. CHAR columns must be **SET** with CHAR expressions, numeric columns must be initialized with numeric data, and DATE fields must be initialized with valid date/time values.

In the second form, the positioned UPDATE modifies one record at the current position of a cursor. The expressions must match the types of the field_name(s). Character string fields must be assigned character string values, and numeric fields must be assigned numeric values. Only floating point fields can be assigned values expressed in exponential notation.

Positioned UPDATE is used only in C programs. The referenced cursor must be open and not at beginning or end of file. That is, at least one row must be FETCHed before the positioned UPDATE can be executed.

A typical application program sequence is to declare a cursor, open the cursor, fetch a record, modify the record, and execute the positioned UPDATE to change the modified data.

Combined Index

This Index also includes references to the **c-tree Plus Programmer's Reference Guide**.

Index

All references to the **c-tree Plus Function Reference Guide** are preceeded by the letter **F**.

A

Abort .....4-6, 11-6 - 11-7, F1-3, F1-16
AddKey 4-2, 4-5, 7-5, F1-4, A-3, A-11, A-13
ADDREC
    See AddRecord
AddRecord 3-8, F1-6, F1-88, F1-161
ADDRES
    See AddResource
AddResource .....4-7, 5-2, 5-6, 5-9, 7-8, 8-3 - 8-4,
    .....F1-8, F1-124
ADDVREC
    See AddVRecord
AddVRecord 5-6, 5-9, F1-10, F1-88, A-8
ALCSET
    See AllocateSet
AllocateSet ... 5-8, 7-12, F1-12, F1-15, F1-19, F1-66
ALTER TABLE (sql statement) F4-3
alternate collating sequence 4-7, 5-19, 12-3,
    ..... F1-33, F1-35, F1-67, F1-173, D-6
    See also GetAlternateSequence
    resource 8-6
    See also SetAlternateSequence
Atomicity
    See transaction ,Atomicity
    See Transaction processing
AUTHORIZATION F4-28
automatic recovery .....3-10, 11-7 - 11-8
    See Transaction processing
    See Transaction, automatic recovery
automatic sequence number 5-21

B

B-Tree
    node locks 9-1
    nodes 5-11
BAT_COMPLETE 8-11
BAT_DEL 8-9
BAT_GET 8-9
BAT_LOK_RED 8-10
BAT_LOK_WRT 8-10
BAT_RET_POS 8-10
BAT_RET_REC 8-10
BAT_VERIFY 8-10
Batched operations
    See DoBatch
BATSET
    See DoBatch
Begin 4-6, 11-3, 11-5, F1-13
branching factor 3-5
buffers
    See Index file, buffers
bufs 5-11

C

ChangeSet 5-8, 7-12, F1-12, F1-15, A-10
char
    See TEXT pointers
CHECKLOCK 3-10, 6-1, 10-2, 11-4
CHECKREAD 3-10, 6-1, 10-2
CHGSET
    See ChangeSet
ClearTranError 4-6, 11-7, F1-16, F1-21, F1-176
Client/Server 1-1
CLIFIL
    See CloseIFile
CLISAM
    See CloseISAM
CLOSE (sql statement) F4-4
CloseFile .....4-3 - 4-4, 7-6, F1-17
CloseIFile 5-5, F1-18, A-2
CloseISAM 5-3, 5-5, 7-9, 10-2,
    .....F1-19, F1-66, F1-95, F1-180

CloseRFile 5-5, 8-6, F1-20
CLRFIL
See CloseRFile
CLSFIL
See CloseFile
Collating Sequence
See Alternate Collating Sequence
Commit .. 4-6, 11-3 - 11-5, 11-7, F1-13, F1-16, F1-21
COMMIT (sql statement) F4-5
Common Entry Point .....E-1 - E-12
compiler
Microsoft C 3-3, 9-2
concurrent access 6-1
CONFIG.SYS 3-9
COUNT parameters 3-1
cpybig 3-3
cpybuf ..... 3-3 - 3-4, F1-23
CREATE INDEX (sql statement) F4-6
CREATE SCHEMA (sql statement) F4-7
CREATE TABLE (sql statement) .... F4-8 - F4-10
CREATE VIEW (sql statement) .... F4-11 - F4-12
CreateDataFile 4-2, 4-4, 5-12,
..... F1-24, F1-27, F1-61, A-15
Superfile creation 8-7
CreateDataFileXtd 12-1, F1-27
CreateIFile 5-2, 5-5, 5-16, 7-10, 8-6,
.....F1-19, F1-28, F1-30
CreateIFileXtd 12-1, F1-30
CreateIndexFile ..... 4-2, 4-4, 5-12 - 5-13,
.....F1-31, F1-34 - F1-35, A-15
CreateIndexFileXtd 12-1, F1-34
CreateIndexMember 4-4, 5-12, F1-35
CreateISAM .... 5-1 - 5-2, 5-5, 5-14, 5-16, 7-7, 10-2,
..... F1-19, F1-37, F1-40, A-2, A-7
CreateISAMXtd 12-1, F1-40, F1-181
CREDAT
See CreateDataFile
CREDATX
See CreateDataFileXtd
CREIDX
See CreateIndexFile
CREIDXX
See CreateIndexFileXtd
CREIFIL
See CreateIFile
CREIFILX
See CreateIFileXtd
CREISAM
See CreateISAM
CREISAMX
See CreateISAMXtd
CREMEM
See CreateIndexMember
ctaerr.h 2-1
ctclib.c 9-2, F1-126
ctclib.d 9-2
CTCMPC F1-46
ctcmpl.h F1-6, F1-115, A-3
ctdecl.h 2-1, 3-4, F1-17
cterrc.h 2-1, 3-13
ctexam.p 5-20, 7-14
ctexmc.c 5-20, 7-14
ctexmg.c 5-20, 6-5, 7-14
ctibld.c F1-151
ctifil.h 2-1, 5-25, 7-9
ctixmg.c .....5-9, 5-20, 7-14 - 7-15
ctoptn.h ..... 2-1, 3-4, 5-12 - 5-14, 5-21, 9-1,
..... F1-29, F1-35, F1-38, F1-96, F1-112,
.... F1-126 - F1-127, F1-132, F1-135, F1-150,
..... A-3 - A-4, A-6 - A-8, A-13 - A-14
ctparm.h E-1
ctport.h ..... 3-1 - 3-2, 12-3, F1-4, F1-110, D-3, D-5
ctrbld.c A-7
ctree function E-1
ctsql.h 15-4
ctsrvr.c F1-49, F1-53, F1-155, F1-157,
.....F1-169, F1-189 - F1-190
ctstdr.h 2-1
ctvxam.p 5-20, 5-22, 7-14
ctvxmg.c 5-9, 5-20, 5-22, 7-14
Current ISAM Record F1-174
See ISAM,Current ISAM record
CurrentFileOffset 5-4, 5-6, F1-42
CurrentISAMKey 5-6, 5-24, F1-43

D

d-tree 1-5, 8-6, D-1, D-5
data buffering 11-8
data buffers F1-93, F1-97, F1-137
data file description record 5-11, 7-7
data record
delete flag 3-7
deletion 6-5
first usable F1-114
position 3-13
data types 3-1

DATENT
See NbrOfRecords
datlen 5-12
datno 5-11
DEAD_ERR 6-4
deadlock 9-1
DECADD F1-4
DECLARE CURSOR (sql statement) F4-13
DELBLD
See DeleteKeyBlind
DELCHK
See DeleteKey
DELETE (sql statement) F4-14
deleted records F1-155, F1-157
DeleteFile 4-4, F1-44
DeleteKey ..... 4-3, 4-5, 7-6, F1-45, A-13 - A-14
DeleteKeyBlind 4-3, 4-5, F1-47, A-1, A-11
DeleteRecord . 3-7, 5-3, 5-6, 7-8, 7-13, F1-49, F1-124
DeleteResource .....4-7, 8-3 - 8-4, F1-51
DeleteVRecord 5-6, F1-53
DELFIL
See DeleteFile
Delimiter F1-178, D-3
See Field delimiter
DELREC
See DeleteRecord
DELRES
See DeleteResource
DELVREC
See DeleteVRecord
descending key segment values 5-18
descending key sequence D-6
destbuf (sql function) 15-6, F3-3
destfil (sql function) 15-6, F3-4
DISABLERES 3-10, 8-3
DLOCK 9-2
DLOK_ERR 6-4
DoBatch .....5-8, 8-8 - 8-11, F1-55 - F1-57, A-11
Partial Key Request 8-9
DODA 8-6, F1-146, D-5
DROP TABLE (sql statement) F4-15
DROP VIEW (sql statement) F4-16
Dummy lock file 3-11, 9-2
DUPCHANEL 3-10, 10-3
duplicate key equality search F1-61, F1-74
duplicate keys F1-5, F1-72, F1-107
dynamic memory usage F1-19, F1-37, F1-91,
..... F1-95, F1-134

E

Embedded SQL
See SQL,Embedded
ENABLE 6-2, 6-4, F1-110
ENABLE_BLK 6-2, 6-4, F1-110
EnableResource 4-7, 8-3, 8-5, F1-59
ENARES
See EnableResource
EQLKEY
See GetKey
EQLREC
See GetRecord
err_pos 14-3
error code 3-13
Error handling 5-3
EstimateKeySpan 4-5, F1-60
ESTKEY
See EstimateKeySpan
exact match of duplicate keys F1-61, F1-74
EXCLUSIVE 3-10, 6-1
extended file functions F1-27, F1-30, F1-34,
..... F1-40, F1-93, F1-97, F1-129 -
.....F1-130, F1-133, F1-137, F1-153
extended function calls
file extensions 5-25
superfiles 8-7
Extended functions 4-3, 5-4, 12-1, F1-181

F

FairCom Server
See Server
FairCom SQL Server
See Server,SQL
FETCH 15-5
FETCH (sql statement) 15-6, F4-17
FETCHP (sql statement) F4-18
Field delimiter 3-9, 5-15, F1-178
field type D-3
File
creation ..... F1-24, F1-27 - F1-28, F1-31,
..... F1-34, F1-37
deletion F1-44
numbering 3-11
File mode 3-8, 3-11, 11-3, 11-5, F1-24
DISABLERES 8-3

multi user 6-1
SUPERFILE 8-7
file numbering 10-4
File passwords
See Security
File Permissions
See Security
File recovery 3-13
See also RebuildIFile
File Server ..... 10-1 - 10-4
filmod ..... 5-12 - 5-13, 6-1
See file mode
filnam ..... 5-11 - 5-12
first usable data record F1-114
FirstInSet ..... 5-8 - 5-9, 7-12, F1-12, ... F1-15, F1-61, F1-74, F1-118, F1-141, A-7
FirstKey 3-12, 4-5, F1-63, A-13
FirstRecord 3-12, 5-6, 7-11, F1-64
FIXED 3-9
fixed record length
See Record,Fixed length
FPUTFGET 9-1, F1-91, F1-108, F1-123
FRCKEY
See KeyAtPercentile
FREE ..... 6-3 - 6-4, F1-110
FreeSet 5-8, F1-66
FRESET
See FreeSet
FRSKEY
See FirstKey
FRSREC
See FirstRecord
FRSSET
See FirstInSet

G

GetAlternateSequence 4-7, 5-19, F1-67
See also alternate collating sequence
GETALTSEQ
See GetAlternateSequence
GETCURK
See CurrentISAMKey
GETCURP
See CurrentFileOffset
GetDODA 4-7, 8-6, F1-68
GETFIL
See GetFileInfo
GetFileInfo 4-4, F1-43, F1-70
GetGTEKey 4-6, 5-9, F1-72
GetGTERecord F1-74
GetGTKey 4-6, 5-9, F1-76
GetGTRecord 5-6, 7-13
GetIFile 5-5, 8-6, F1-77
GetKey ..... 4-2 - 4-3, 4-5, 7-5, F1-72, F1-78
GetLTEKey 4-6, F1-79
GetLTERecord 5-6, F1-80
GetLTKey 4-6, 5-9, F1-82
GETNAM
See GetSymbolicNames
GetRecord 5-2, 5-6, 7-8, F1-74, F1-83
GETRES
See GetResource
GetResource ..... 4-7, 8-3 - 8-4, F1-85
GetSerialNbr F1-88
GetSymbolicNames 4-4, F1-89
GetTempFileName F1-90
GETVLEN
See VRecordLength
Group ID
See Security
GTEKEY
See GetGTEKey
GTEREC
See GetGTERecord
GTKEY
See GetGTKey
GTVLEN
See VDataLength
GUEST
See Security

H

Header record 3-13

I

I/O Management 10-3
IDXENT
See NbrOfKeyEntries
IFIL structure 3-2, 5-25, 7-9
IGRANT (sql statement) ..... F4-19 - F4-21
IIDX Structure 5-26, 7-10
INCADD F1-4
Include statements 2-1

Incremental ISAM .....F1-18, F1-20, F1-28, ... F1-37 - F1-38, F1-95, F1-127, ... F1-131, F1-134 - F1-135, F1-150
creation F1-30
include 5-2
See ISAM, Incremental ISAM structure
Resources 5-29
Index file
buffers 3-7, 5-11, 7-3
description record 5-12, 7-7
index member 3-11, F1-31
index member record 5-14
InitCTree ..... 3-12, 4-1, 4-4, 7-3, 10-2, 15-1, ... F1-31, F1-91, F1-93, A-2 - A-3, A-7
InitCTreeXtd 12-1, F1-93
initialization record 5-11, 7-7
InitISAM ...5-5, 7-10, 10-2, F1-95, F1-97, A-2, A-7
InitISAMXtd 12-1, F1-97, F1-181
INSERT INTO (sql statement) ..... F4-22 - F4-23
Integer
four byte 3-2
signed 3-1
unsigned 3-2
INTISAM
See InitISAM
INTISAMX
See InitISAMXtd
INTREE
See InitCTree
INTREEX
See InitCTreeXtd
Isam
record rewrite 5-3
creation F1-40
Current ISAM record 5-3, 7-13
Data record locks 6-2
Errors 5-3
file relationships 5-9
functions .....3-12, 5-1 - 5-29
Incremental 2-1
Incremental ISAM 5-5
Incremental ISAM structure ..... 3-6, 5-8, ...5-24 - 5-29, 7-9, 7-14 - 7-15
Incremental ISAM, resources 8-6
logging off server 10-2
logging on to Server 10-2
parameter file ..... 3-6, 5-10 - 5-15, 7-6
Parameters .....5-1, 5-8 - 5-9
ISAM, Current ISAM record .....F1-158, F1-165, ...F1-168, F1-170
isam.c 5-1
isam_err 3-13, 5-3, 7-8, F1-1
isam_fil 3-13, 5-3, 7-8, F1-1
ISEG Structure 5-27, 7-10
ITIM_ERR 6-4

K

Key
key value padding 7-4
translation 7-4
key segment
parameter 5-1, 5-9
descending values 5-18
description record 5-14, 7-8
key value padding F1-5, F1-107
key value sets F1-12, F1-61
key value suffix F1-5, F1-107
KeyAtPercentile 4-5, F1-60, F1-99
keydup 5-12, 5-14
keylen 5-12, 5-14
keyno 5-12, 5-14
Keys ..... 3-4 - 3-6
compressed ..... 3-4 - 3-6
duplicate 3-6
fixed length ..... 3-4 - 3-5
ISAM 3-6
key length 3-6
key segment parameter 3-12
key types 3-4
key value assembly 5-23
key value padding 3-4, 4-2, 5-9
key value suffix 3-6
segment length 5-15
segment mode ..... 5-16 - 5-23, D-1
segment position 5-15
sequence number 5-18
target key value 5-24
keytyp 5-12, 5-14

L

LAN 1-2, F1-152
LastInSet .....5-8 - 5-9, F1-12, F1-15, F1-80, ... F1-101, A-7
LastKey 4-6, F1-103, A-13

LastRecord 5-6, F1-104
Resources 8-3
LKISAM
See LockISAM
LoadKey 4-6, F1-106, A-4
local file buffering 10-1
LOCK 9-2
LockData ..... 4-5, 6-3 - 6-4, 7-5, 11-4,
.....F1-21, F1-86, F1-108, F1-111
Locking
File Locking ..... 6-1 - 6-3
ISAM data record locks 6-2
See Record locking
LockISAM .....5-6, 6-2, 7-8, 8-10, 11-4 - 11-5,
..... F1-13, F1-21, F1-86, F1-110, A-7
transactions 11-3
Logging off Server 4-4, 5-5, 10-2, F1-180
Logging on Server 4-4, 5-5, 10-2
LOKREC
See LockData
LONG values 3-2
Low Level Functions .....3-11, 4-1 - 4-7
lowlevel.c 4-1
LSTKEY
See LastKey
LSTREC
See LastRecord
LSTSET
See LastInSet
LTEKEY
See GetLTEKey
LTEREC
See GetLTERecord
LTKEY
See GetLTKey

M

MAX_DAT_KEY F1-29, F1-38,
.....F1-128, F1-132, F1-135, F1-151
MAX_KEY_SEG F1-29, F1-38, F1-128,
.....F1-132, F1-135, F1-151
MAX_LOCKS F1-112
MAXFIL F1-38, F1-96, F1-135
MAXLEN ..... F1-35, F1-38, F1-126 - F1-127,
.....F1-132, F1-135, F1-150
Members F1-35
Microsoft C
See Compiler, Microsoft C
MIDSET
See PositionSet
minimum record length 3-9
Multi user 1-3, 3-10, 7-3, 9-1
interference 6-4
multi tasking 6-1, 10-1
multi-tasking 3-10
updates 6-5
Multiple indices 7-14
multiple sets F1-12

N

NbrOfKeyEntries 4-6, F1-123
NbrOfRecords 4-5, F1-124
Network 3-10, 6-1, 10-1
See also LAN
NewData ...3-8, 4-2, 4-5, 7-5, F1-88, F1-114, F1-124
NEWREC
See NewData
NewVData 4-5, F1-88, F1-116, F1-157
NEWVREC
See NewVData
NextInSet 5-8, 7-12, F1-12, F1-15, F1-118, A-7
NextKey 3-7, 4-6, F1-120
NextRecord 5-6, 7-11, 7-13, F1-121
node locks 9-1
node sectors 5-11
nomemb 5-12
null
character 3-3
null byte 3-4, 5-9
null key flag 5-13, 7-7
NXTKEY
See NextKey
NXTREC
See NextRecord
NXTSET
See NextInSet

O

OPEN (sql statement) F4-24
OpenFile 3-12, 4-2, 4-4, 5-11,
..... F1-31, F1-125, F1-130, A-15
Superfile 8-7

OpenFileWithResource 5-2, 5-5, 8-6, 12-1,
..... F1-127, F1-129
OpenFileWithResources F1-28
OpenFileWithResourceXtd F1-129
OpenFileXtd 12-1, F1-130
OpenIFile 5-2, 5-5, 5-11, 5-16, 7-10, F1-19,
..... F1-131, F1-133, A-2, A-7 - A-8
OpenIFileXtd 12-1, F1-133
OpenISAM .....3-12, 5-1 - 5-2, 5-5, 5-14, 5-16, 7-7,
..... 10-2, F1-19, F1-134, F1-137, A-2, A-7
OpenISAMXtd 12-1, F1-137, F1-181
OPNFIL
See OpenFile
OPNFILX
See OpenFileXtd
OPNIFILX
See OpenIFileXtd
OPNISAM
See OpenISAM
OPNISAMX
See OpenISAMXtd
OPNRFIL
See OpenFileWithResource
OPNRFILX
See OpenFileWithResourceXtd

P

Padding 5-16, F1-178
Set padding value 3-5
Partial Key Request 8-9
Passwords
See Security
PCNET 7-1
Performance
buffering 3-10, 11-8
disk head movement 3-5
multiple channels 10-3
PERMANENT 3-8
Permanent file open 3-8
Permission mask
See Security
Permissions
See Security
PKEYREQ 8-9, F1-55
PositionSet 5-8, F1-139
PREIMG ..... 3-10, 10-2, 11-4 - 11-5
preprocessor
See SQL,preprocessor
PreviousInSet 5-8, F1-12, F1-15, F1-141, A-7
PreviousKey 4-6, F1-143
PreviousRecord 5-6, 7-13, F1-144
Resources 8-3
productivity
message traffic 10-1
PROTOTYPE 3-4
Prototyping 3-3
PRVKEY
See PreviousKey
PRVREC
See PreviousRecord
PRVSET
See PreviousInSet
PutDODA 4-7, 8-6, F1-146, D-1, D-5
PUTFIL
See UpdateFileMode

R

r-tree ..... 1-5, 5-11 - 5-14, 5-21, 7-7,
..... F1-89, D-5 - D-6
DODA 5-13
symbolic index name 5-13
r_fetch (sql function) F3-5
r_insert (sql function) F3-6
r_update (SQL function) F3-7
RBLIFILX
See RebuildIFileXtd
RDVREC
See ReadVData
read only access 6-1
ReadData 3-12, 4-3, 4-5, 7-6, F1-147, A-3
READFIL 3-10, 6-1
READREC ..... 6-2 - 6-4, F1-110
ReadVData 4-5, F1-148
Rebuild .....5-12 - 5-13, 7-6, 7-14 - 7-15
function F1-150, F1-153
RebuildIFile 3-13, 5-5, F1-150, F1-153
RebuildIFileXtd 12-1, F1-153
Record
deletion 6-5
fixed length 3-7, 3-9
minimum length 3-9
record length 3-7
relative record number 3-7

rewrite 6-5
update 6-5
Record locking ..... 6-1 - 6-3
Record Schema ..... 3-9, 5-15, F1-146, D-1 - D-2
record schemas 12-3
REDREC
See ReadData
REDVREC
See ReReadVRecord
REGADD F1-4
Relational database 13-1
ReleaseData 3-7, 4-3, 4-5, 7-6,
..... F1-114, F1-124, F1-155
ReleaseVData 4-5, F1-157
ReReadRecord 5-7, F1-158, A-3
ReReadVRecord 5-6, F1-61, F1-64, F1-74,
..... F1-80, F1-83, F1-101, F1-104, F1-118,
..... F1-121, F1-141, F1-159, A-3, A-8 - A-9
RESET 6-3, F1-110
ResetRecord 5-4, 5-7, F1-161
Resource . . 4-7, 8-1 - 8-5, F1-85, F1-104, F1-144, D-4
Alternate Collating Sequence . . F1-67, F1-173
disable 8-3
DODA 4-7
Enable after creation F1-59
file mode 3-8
Functions 8-3
IFILE open F1-127
IFIL structure F1-28
Incremental ISAM 5-5, 5-29, 8-6
Incremental ISAM close F1-20
LastRecord 8-3
locking 6-3
mode 3-10
Name 8-2
Number 8-2
PreviousRecord 8-3
record marker 3-8
record schema F1-68, F1-146, D-1
Type 8-2
RESTORE 6-3, F1-110
RESTORE_BLK 6-3, F1-110
RestoreSavePoint ..... 4-6, 11-6 - 11-7,
..... F1-16, F1-21, F1-163
RESTRED 6-3, F1-110
RETREC
See ReleaseData
RETVREC
See ReleaseVData
reusing deleted records F1-155, F1-157
REVOKE (sql statement) F4-25
rewrite
See record,rewrite
ReWriteRecord 5-3, 5-7, 5-18, 7-13,
..... F1-161, F1-165, A-1
ReWriteVRecord 5-3, 5-7, F1-168, A-1
RLOCK 9-2
ROLLBACK WORK (sql statement) F4-26
Rows
See SQL,Rows
RRDREC
See ReReadRecord
RTREE A-6
RWTREC
See ReWriteRecord
RWTVREC
See ReWriteVRecord

S

savepoints
See Transaction,savepoints
sect 5-11
security ..... 4-3 - 4-4, 5-4, 12-1 - 12-4
File passwords 12-4
File Permissions 12-2
function F1-171
Groups 12-2
GUEST group 12-2
modify permissions F1-171
Passwords 12-2
Permission mask 12-3
permission masks F1-27, F1-30,
..... F1-34, F1-40, F1-153
resource 8-6
User ID 12-2, F1-40, F1-93, F1-97, F1-137
user password F1-40, F1-93, F1-97, F1-137
user profile mask . F1-40, F1-93, F1-97, F1-137
SELECT (sql statement) F4-27
SELECT INTO 15-5
Sequence number 3-7
serial number . . 5-18, F1-88, F1-114, F1-116, F1-168
See also GetSerialNbr
Serializability, ANSI feature 14-3
SERIALNUM
See GetSerialNbr
Server ..... 1-1 - 1-2, 1-5, 3-8, 3-10 - 3-11, F1-32

logoff 4-4, 5-5, 10-2, F1-180
logon 4-4, 5-5, 10-2
security 12-1
server name F1-40, F1-93, F1-97, F1-137
server-application relationship 10-1
SQL .....1-1, 1-5, 13-1 - 13-2, D-1
system administrator 12-2
SET AUTHORIZATION (sql statement) ...F4-28
Set Functions 5-7, 7-11, F1-12, F1-15, F1-61,
.....F1-66, F1-101, F1-118, F1-139, F1-141
SetAlternateSequence 4-7, 5-19, F1-33, F1-35,
..... F1-173, D-7
See also alternate collating sequence
SETALTSEQ
See SetAlternateSequence
SETCURI
See SetRecord
setdest (sql function) 15-6, F3-8
SetRecord 5-4, 5-7, F1-174
sets of key values F1-12, F1-61
SetSavePoint .....4-6, 11-6 - 11-7, F1-176
SETVARBYTS
See SetVariableBytes
SetVariableBytes 3-9, 4-7, 5-15, 8-6,
.....F1-178, D-3, D-6
setvars 15-5
setvars (sql) F3-9
SHARED 3-10, 6-1, 7-3
single entry point F1-181
Single user mode 7-3
Non-Server 1-3
Server 1-3
SORT_SPACE F1-151
SQL
Columns 14-1
Embedded ..... 1-3, 13-1, 15-1 - 15-3, F3-1
embedded SQL 13-1
Error codes .....14-3 - 14-4, B-1 - B-10
interactive 13-1
interactive SQL 13-1
preprocessor 15-1
record schema D-1
rows 14-1
Select statement .....F2-1 - F2-22
See also Server,SQL
Tables 14-1
view 14-1
Views 14-1
SQL preprocessor .....F4-17 - F4-18
sql1 (sql function) 15-5, F3-10
sql2 (sql function) 15-5, F3-11
sql_err 14-3
StopUser 4-4, 7-4, 7-9, 15-1, F1-180
STPUSR
See StopUser
suffix F1-5, F1-107
Superfile .....3-10, 8-6 - 8-7, F1-25, F1-32,
.....F1-64, F1-104, F1-121, F1-125, F1-144
file mode 3-8
Members 8-7
NewData F1-114
SQL 14-1
SUSPEND 6-3, F1-110
SWTCURI 5-4
symbolic data field name 5-12
symbolic index name 5-13
System Administrator 12-4

T

Tables
See SQL,Tables
target key value F1-181
Temporary File Name F1-90
TEXT pointers 3-2
TFRMKEY
See TransformKey
TMPNAME
See GetTempFileName
TRANABT
See Abort
TRANBEG
See Begin
TRANCLR
See ClearTranError
TRANEND
See Commit
TRANRST
See RestoreSavePoint
Transaction 1-3, 3-8, 3-10
abort 11-6
Atomicity 11-2
automatic recovery 11-2
commit 11-5
locking F1-111
mode 10-2
number 11-5

processing . 5-12, 10-3, 11-1 - 11-8, F1-6, F1-10
savepoints 11-6
SQL transaction processing 14-2
TRANSAV
See SetSavePoint
TransformKey .....5-7, 5-18, 5-24, 7-8, 7-12 - 7-13, ...F1-40, F1-93, F1-97, F1-137, F1-181, E-2
TRNLOG 3-10, 10-2, 11-3, 11-8
TRNLOG ..... 11-4 - 11-5
two-phase locking protocol 6-2, F1-108, F1-111

U

UCOUNT parameters 3-2
UDLOCK 9-2
uerr_cod 3-13, 4-3, 7-4, 14-4, F1-1
UINT parameters 3-2
UNIX 1-2
UNLOCK 9-2
UPDATE (sql statement) F4-29
update record 6-5
UpdateFileMode 4-4, F1-183
UpdateResource .....4-7, 8-3 - 8-4, F1-185
UPDCURI
See ResetRecord
UPDRES
See UpdateResource
User ID
See Security
user profile mask . 5-18, F1-40, F1-93, F1-97, F1-137
USERPRF_NTKEY ...F1-40, F1-93, F1-97, F1-137

V

variable length
data records 5-15, 7-14
records 3-9
variable length record .....F1-116, F1-148, F1-157, ... F1-159, F1-168, F1-187 - F1-188, F1-190
key segment 5-18
variable length records
batch processing F1-57
varying length field 5-10
VDataLength F1-148, F1-187
Views
See SQL,Views
VIRTUAL 3-8
Virtual file open 3-8
VLENGTH 3-9
VOID 3-2
VRecordLength 4-5, 5-6, F1-159, F1-188
VRLEN parameters 3-2

W

white space 5-11
WriteData 4-2, 4-5, 7-5, F1-189, A-3
WRITETHRU 3-10, 11-8
WriteVData 4-5, F1-190, A-3
WRTREC
See WriteData
WRTVREC
See WriteVData

X

XENIX
See UNIX
xtdsiz .....5-12 - 5-13, 7-3

Z

zero byte 3-4

Reader's Review

The **FairCom** Documentation Department wants to provide manuals that meet the needs of all of our users. After reading and using this manual, please take a few moments to answer the following questions. Your input can help us more fully meet your needs in the development of future documentation. Your name, company, and phone number are optional, but will help us to serve you better.

Manual Name	*Your Company Name*
Operating Environment you are programming in	*Your Telephone Number*
Your Name	*Type of products you are developing*

Please rate this manual in the following areas:	*10 = Highest 0 = Lowest*
Clarity of instructions	
Completeness of information	
Logical organization	
Easy to read	
Ease of finding information	
Usefulness of examples	
Usefulness of reference material *(Table of Contents, Index, Appendices)*	

Please continue on the back of this page

Are there areas of the manual that you found especially helpful? If so, what are they?

Are there other areas of the manual that you feel could be improved? If so, how?

Are there any particular concepts that show be explained in more detail?

Was there any specific information that you had a difficult time finding?

Please send these or any other comments/suggestions to the following address:

FairCom Corporation
Documentation Department
4006 West Broadway
Columbia, MO 65203

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

c-tree Plus™

Function Reference Guide

FAIRCOM
corporation

4006 WEST BROADWAY
COLUMBIA, MO 65203
PHONE 314.445.6833
800.234.8180
FAX 314.445.9698